Probabilistic Verification of Network Configurations

Samuel Steffen
ETH Zurich, Switzerland
samuel.steffen@inf.ethz.ch

Timon Gehr
ETH Zurich, Switzerland
timon.gehr@inf.ethz.ch

Petar Tsankov
ETH Zurich, Switzerland
petar.tsankov@inf.ethz.ch

Laurent Vanbever
ETH Zurich, Switzerland
lvanbever@ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

ABSTRACT

Not all important network properties need to be enforced all the time. Often, what matters instead is the fraction of time / probability these properties hold. Computing the probability of a property in a network relying on complex inter-dependent routing protocols is challenging and requires determining all failure scenarios for which the property is violated. Doing so at scale and accurately goes beyond the capabilities of current network analyzers.

In this paper, we introduce NetDice, the first scalable and accurate probabilistic network configuration analyzer supporting BGP, OSPF, ECMP, and static routes. Our key contribution is an inference algorithm to efficiently explore the space of failure scenarios. More specifically, given a network configuration and a property ϕ, our algorithm automatically identifies a set of links whose failure is provably guaranteed not to change whether ϕ holds. By pruning these failure scenarios, NetDice manages to accurately approximate $P(\phi)$. NetDice supports practical properties and expressive failure models including correlated link failures.

We implement NetDice and evaluate it on realistic configurations. NetDice is practical: it can precisely verify probabilistic properties in few minutes, even in large networks.

CCS CONCEPTS

• **Mathematics of computing → Probabilistic inference problems**; • **Networks → Network properties**.

KEYWORDS

Network analysis, Failures, Probabilistic inference, Cold edges

ACM Reference Format:
Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3387514.3405900

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405900

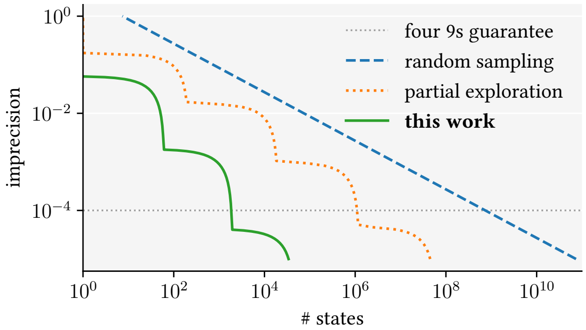


Figure 1: Comparison of approaches for probabilistic network analysis in a network with 191 links and link failure probability 0.001. The confidence for sampling (Hoeffding's inequality) is $\alpha = 0.95$.

1 INTRODUCTION

Ensuring network correctness is an important problem that has received increased attention [1, 4, 12, 16, 19, 31, 40]. So far, existing approaches have focused on verifying "hard" properties, producing a binary answer of whether the property holds under all or a fixed set of failure scenarios.

Besides hard properties, network operators often need to reason about "soft" properties[1] which *can* be violated for a small fraction of time (e.g. 0.01%). Among others, allowing properties to be violated allows for cheaper network designs, e.g. by reducing over-provisioning. Soft properties typically emerge when reasoning about compliance with Service Level Agreements (SLAs). SLAs can be defined with respect to any metric (e.g. path availability, average hop count, capacity) and are traditionally measured in "nines": For instance, an IP VPN provider might guarantee internal path availability between its customers for 99.999% (five 9s) of the time, and two-path availability for 99.99% (four 9s).

Similarly to verifying hard properties, computing the probability of a soft property requires analyzing the network forwarding behavior emerging from a network configuration (i.e. the network control plane) in many, possibly all, environments (e.g. failure scenarios). A key difference is that verifying a hard property aims at checking the absence of a counter-example (e.g. a failure scenario in which the property is violated), not at computing *how many*

[1] This need is exemplified by a survey we conducted amongst network operators (52 answers). In this survey, 94% of operators indicated that they care about probabilistic network analysis. At the same time, 83% of them indicated that it is currently difficult to do so. See App. A.1 for details.

such counter-examples exist and *how likely* they are. This is an important distinction as it enables modern network verification tools such as Minesweeper [4], BagPipe [40], ARC [19], Tiramisu [1], and ERA [12], to reason abstractly about many environments.

Challenge Modern networks often rely on a combination of BGP and IGPs for their routing decisions. Exactly computing how frequently a property holds in such a network is challenging, unless its size is tiny enough to admit brute forcing all failure scenarios. Instead, one can aim at *approximating* the probability by exploring a subset of failure scenarios. As we show though, naive partial exploration often leads to inaccurate approximations, making it impossible to verify SLA properties that must hold almost all the time, e.g. 99.99% or more. Since violating SLAs may lead to monetary sanctions, being accurate is key.

We illustrate the challenges of accurately approximating these probabilities with two intuitive strategies: (i) *partial exploration*, in which one only considers up to k link/node failures; and (ii) probability estimation via *sampling*. For each, we explore the relationship between the achieved precision and the number of explored states (see Fig. 1, note the log-log scale). The dotted line indicates an imprecision of 10^{-4}, necessary to verify a four 9s SLA.

Attempt 1: Partial exploration This strategy explores the failure scenarios in decreasing order of likelihood, i.e. the most likely first. Due to the exponential decay of probabilities though, this strategy must still explore many states to reach an accurate approximation (except in small networks). For example, consider the Colt network [27] (191 links) with a link failure probability of 0.001 [20]. As visualized by the dotted line in Fig. 1, one needs to explore more than 10^6 states, almost all scenarios with up to 3 links failures, to reach the precision required to verify a four 9s SLA. [2] Clearly, this does not scale. As an illustration, Batfish [16] requires around 30 sec on average to analyze BGP configurations for *one* scenario of a similarly sized network. The fundamental issue is that the gain in precision becomes exponentially smaller as more failures are considered: In Colt, the probability of at most 1 link failing is $0.999^{191} + 191 \cdot (0.999^{190} \cdot 0.001) \approx 0.984$, while the probability of at most 2 (resp. 3) links failing is ≈ 0.9990 (resp. ≈ 0.9999).

Attempt 2: Sampling This strategy samples k scenarios from the failure distribution (by independently sampling each link's failure status), checks the property for each of them, and computes the fraction of samples for which the property holds. Technically, this corresponds to maximum likelihood estimation (MLE) of the property probability. We compute the precision of the estimate using Hoeffding's inequality [23]. While the precision is independent of the network size, this strategy needs an intractable number of samples to achieve high precision. We illustrate this in Fig. 1 (see the dashed line, we use confidence $\alpha = 0.95$). One needs to sample 10^9 states in order to reach the precision required to verify a four 9s SLA, which clearly does not scale.

We note that there exists a variety of more advanced sampling strategies such as importance sampling, Markov Chain Monte Carlo, or Metropolis-Hastings (see e.g. [5, Chapter 11]). However, these are typically concerned with low-probability observations (which is irrelevant in our context) and may produce incorrect results due to weak theoretical guarantees [11]. While there also exist alternative concentration bounds for MLE such as the Wald interval [28], their empirical guarantees can be violated in practice [7].

To sum up, a fundamental research question is still open: *Is it possible to build a scalable network verification tool that can precisely verify probabilistic properties to support even stringent SLAs (four/five 9s) in large networks relying on BGP and common IGPs?*

NetDice: Scalable and precise probabilistic analysis We answer positively by presenting the probabilistic configuration analyzer *NetDice*. Given a network configuration, the external routing inputs (if any), a probabilistic failure model, and a property ϕ, NetDice accurately computes $P(\phi)$, the probability of ϕ holding.

To precisely and scalably compute $P(\phi)$, NetDice relies on an efficient inference algorithm which is based on two key insights. First, we identify a class of practical properties amenable to probabilistic inference. Second, we show how to leverage these properties' structure to heavily prune the space of failures. More specifically, given a property ϕ, we show how to automatically identify a set of links whose failure is provably guaranteed *not* to change whether ϕ holds. We describe efficient techniques to identify these links for the most common routing protocols (OSPF, BGP) and mechanisms (static routes, route reflection, and load balancing). We also discuss how to extend NetDice to support more protocols.

NetDice reasons probabilistically about *internal* failures, not *external* routing announcements (which we assume are fixed). Concretely, NetDice cannot reason about the uncertainty created by external BGP routes (dis)appearing at the network edge. While this is a limitation, we stress that reasoning about internal failures is already practically relevant (e.g. to guarantee internal SLAs). Further, while learning a precise internal failure model is possible, e.g., from historical data [6, 20, 38], this is not the case for external announcements.

We fully implemented NetDice and published it on GitHub. [3] Our evaluation shows that NetDice can verify practical properties in few minutes, with imprecision below 10^{-4}, even in networks with hundreds of nodes.

Main contributions Our main contributions are:

- An introduction to the problem of probabilistic network analysis—a new facet to the area of network verification (§2).
- A class of practical network properties that are amenable to efficient probabilistic inference (§3).
- A scalable inference algorithm effectively pruning the space of failures for BGP and common IGPs (§4–§5) to compute the probability of a property for an expressive failure model (§6–§7).
- NetDice, an implementation of our approach, evaluated on real network topologies and configurations (§8).

[2] The peculiar shape of the line is due to the log-log scale. Each "bump" in the plot comprises all scenarios with exactly k failures for a specific k. With independent link failures, each such scenario is equally likely, resulting in a linear relationship that manifests as a curved line. However, increasing k leads to exponentially less likely scenarios, resulting in several bumps (one for each k).

[3] https://github.com/nsg-ethz/netdice

2 OVERVIEW

We now describe NetDice on a simple example: consider an OSPF network with 6 routers and link weights as shown in Fig. 2. Assume the operator requests the probability of a waypoint property ϕ. Specifically, she wants to know the probability that traffic from D to C traverses a monitoring node E, given that links may fail probabilistically. While the property ϕ is not a hard requirement, she would like it to be violated only a small fraction of the time.

Input and output NetDice takes four inputs: (i) a network configuration comprising the topology, and the BGP and IGP setup (including external peers, route reflectors, link weights, and static routes); (ii) external BGP announcements; (iii) a failure model specifying the probabilities of (possibly dependent) failures; and (iv) a property ϕ. NetDice returns the probability $P(\phi)$ of the property holding. In our example, we consider shortest path routing (we discuss static routes and BGP in §5) and independent link failures with probability 0.2 (we discuss more complex failure models in §6).

Failure exploration All failures in NetDice are modeled as link failures. For example, a node failure is modeled as a simultaneous failure of all adjacent links (see §6 for details). NetDice therefore computes $P(\phi)$ by exploring the space of possible link failures. Intuitively, it checks ϕ for all possible failure scenarios and sums up the probabilities of those where ϕ holds.

In Fig. 2, we show several link states that NetDice explores, where we depict failed links with a cross. NetDice starts with L_1, the state where all links are up, and then explores L_2, L_3, etc.

Pruning the failure space using cold edges Our key insight is that for any given link state, there may exist links (we call these *cold edges*) whose failure is *guaranteed not to change whether ϕ holds or not.* That is, the state of cold edges is irrelevant for the property at the given link state.

In state L_1 where all links are up (see Fig. 2), the forwarding path from D to C is D–E–C, and therefore our property ϕ holds. All 5 edges outside the shortest path (highlighted in blue) are cold, because any combination of them failing will not change the forwarding path from D to C (and thus the satisfaction of ϕ). There are $2^5 - 1 = 31$ such combinations (we depict three examples in the top right of Fig. 2). NetDice leverages this observation to cover *all* 32 states ($L_1 + 31$) in one go by computing their total probability mass, which is identical to the probability that links D–E and E–C are both up ($p_1 = 0.8^2 = 0.64$). Note that this is significantly more than the probability mass of L_1 alone ($0.8^7 \approx 0.21$).

Determining cold edges for shortest path routing is straightforward. However, extending this idea to real-world networks, which use multiple protocols including BGP, static routes, and shortest paths in combination, is challenging. To this end, in §5, we define algorithms that determine cold edges for any combination of these protocols and formally prove their correctness.

Recursively adding failures Next, NetDice introduces single failures to all non-cold edges, arriving at the states L_2 and L_3. NetDice continues with L_2 and recursively explores all failure scenarios where link D–E fails. Therefore, when processing the subtree at L_3, NetDice never introduces a failure to the link D–E since all such scenarios are already explored. That is, NetDice "locks" link D–E to remain up. The recursion ends when all non-locked links are cold, such as in L_6 where D is disconnected from C.

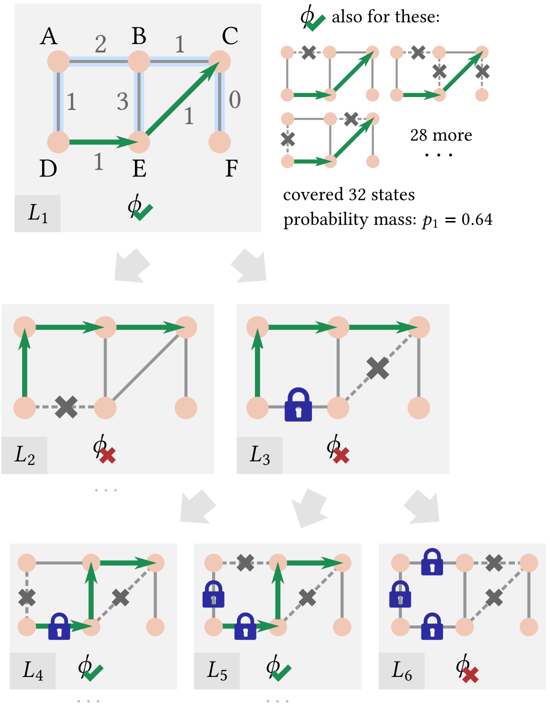


Figure 2: NetDice's failure exploration computing the probability of the property ϕ "traffic from D towards C traverses the waypoint E" under shortest path routing.

For each visited state L_i, NetDice constructs the forwarding graph, determines whether ϕ holds, and computes the covered probability mass p_i (similarly as for L_1). When exploration terminates, NetDice sums the values p_i for which ϕ holds to obtain $P(\phi)$.

For our example, NetDice determines $P(\phi) \approx 0.6932$ by visiting only 15 states. In contrast, a brute force algorithm would need to visit $2^7 = 128$ states. In practical networks with hundreds of links, the benefit of NetDice is even more pronounced (see §8).

We note that while the set of cold edges in each state depends on the flow involved in the property ϕ, it does *not* depend on whether ϕ holds. In particular, NetDice would explore the same states if the waypoint under consideration was A instead of E.

Bounded exploration To speed up inference, NetDice can be configured to stop exploration when a desired level of precision is reached. In this case, the output of NetDice is a probability interval I_ϕ *guaranteed* to contain $P(\phi)$. In practice, very high precision can be obtained by visiting only few states.

3 NETWORK MODEL AND PROPERTIES

We now define the control and forwarding plane model as well as the properties supported by NetDice. We then formally define our problem statement.

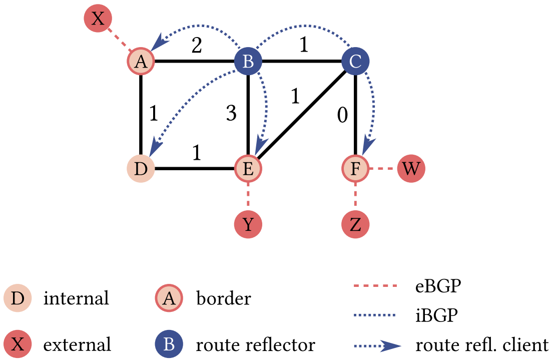


Figure 3: Example topology and BGP session layout.

3.1 Network Model

The internal topology is an undirected weighted graph $G = (V, E, w)$, where the nodes V represent routers, and edges E represent full-duplex links. The function $w\colon E \rightarrow \mathbb{N} \times \mathbb{N}$ assigns to each link a pair of non-negative weights (one weight per direction) to be used for shortest path routing.

Example Fig. 3 shows a simple topology which will serve as a running example throughout this paper. It consists of 6 nodes A–F (i.e., $V = \{\mathrm{A}, \ldots, \mathrm{F}\}$), which are connected by weighted links as indicated. For simplicity, we use the same weights for both directions (e.g., $w((\mathrm{B}, \mathrm{E})) = (3, 3)$).

3.2 Control Plane

Shortest paths and static routes For internal destinations, NetDice supports static routes and shortest path routing protocols such as OSPF and IS-IS, which are modeled by shortest paths under weights w. The network may use equal-cost multi-path (ECMP) forwarding, where all shortest paths are used. The *IGP cost* between two nodes is the sum of weights along a shortest path.

For example, the shortest path between nodes D and C is D–E–C, and the IGP cost between them is 2 (see Fig. 3).

BGP We assume that the network uses BGP to propagate external routing information within the network. The administrator can configure iBGP sessions between nodes, where some nodes may be configured as *route reflectors*. The *border routers* maintain eBGP sessions with external nodes in different ASes. Nodes which are neither external nor border routers nor route reflectors are called *internal nodes*.

For example, in Fig. 3, the border router F maintains two eBGP sessions with external nodes W and Z (the *external peers* of F). Nodes B and C are connected by an iBGP session (i.e., they are *internal peers*) and serve as route reflectors. Node B has three route reflector clients: A, D, and E. Node D is the only internal node in the example.

BGP routing decisions are based on announcements received from external nodes. The BGP protocol, which we will discuss in more detail in §4, determines for each node and destination the *selected next hop*, which is the exit point used by traffic to leave the network. In our model, the selected next hop of internal nodes and route reflectors is always a border router. A border router may select an external peer as the next hop to forward traffic outside of the network.

We make two main assumptions on the BGP configuration of the network. First, similarly to [14, 15, 22], we assume that BGP announcements are not modified internally. Previous measurement studies [10, 39] show that this assumption holds for the vast majority of the networks (except for large transit providers). Second, we assume that BGP converges to a unique state, which is also true in most practical cases [13, 34].

3.3 Forwarding Plane

The combination of BGP, shortest paths, and static routes determines how a node forwards traffic. If a static route is configured, the node uses it, otherwise, the node resolves the next router along the shortest path towards the selected next hop. The forwarding information base (FIB) comprises all forwarding rules for all prefixes at all nodes.

3.4 Properties

In the most general case, a network (forwarding) property is an arbitrary predicate over the FIB determined by the routing protocols described above. While arbitrary properties are very expressive, analyzing their satisfaction under failures is very hard (we will discuss failures in §3.5). For instance, the property may check *each* FIB entry for all nodes and all destinations, which is likely affected by *any* failure. In general, analyzing such properties under failures requires exhaustive enumeration of all possible failure scenarios.

Yet, many practical properties are relatively lightweight in terms of FIB entries they depend on. For instance, an operator may be interested in: (i) whether traffic from a customer for some destination traverses a firewall, which is independent of the FIB entries at routers not receiving that customer's traffic, or (ii) whether the inbound traffic destined to the top-k prefixes share any under-provisioned links, which is independent of FIB entries for other destinations. We capture properties that do not depend on all FIB entries using the concepts of single- and multi-flow properties.

Single-flow properties Let a *flow* be a pair of ingress node and destination prefix. Note that this is *not* equivalent to a TCP flow: there may be many TCP flows between the same ingress and prefix. A property is a *single-flow* property if it only depends on the forwarding graph[4] of a single flow, and is deterministic given this graph (this is, failures are the only source of randomness). Property (i) above is such an example.

Multi-flow properties As a generalization, a *multi-flow* property ϕ only depends on the forwarding graphs of a bounded set $\mathit{flows}(\phi)$ of flows. Such properties are particularly useful to analyze behavior related to congestion or isolation. The property (ii) above is such an example. Note that this class is very general: most forwarding properties can be expressed as a multi-flow property by including *every* flow of the network in the property (resulting in an arbitrary predicate over the FIB of all nodes). Further, note that every single-flow property is also a multi-flow property.

[4] Due to ECMP, there may be multiple paths towards a destination. Hence, the forwarding behavior is described by a general directed graph.

	Property	Meaning
single-flow	$\text{Reachable}_{u \to d}$	Traffic $u \to d$ reaches its destination d.
	$\text{Waypoint}_{u \to d}(w)$	Traffic $u \to d$ traverses node w.
	$\text{Egress}_{u \to d}(e)$	Traffic $u \to d$ leaves network at egress e.
	$\text{PathLength}_{u \to d}(l)$	Traffic $u \to d$ traverses l links.
multi-flow	$\text{Balanced}_F(\mathcal{L}, \Delta)$	Under flows F with given volumes, the load on the links in $\mathcal{L}$ differs by at most Δ.
	Isolation_F	Set of flows F does not share any links.
	$\text{Congestion}_F(l, t)$	Flows F with given volumes together do not exceed volume threshold t for link l.

Table 1: Examples of single- and multi-flow properties supported by NetDice.

Practical few-flow properties NetDice specifically targets "few-flow" properties ϕ, where the size of flows(ϕ) is small. In fact, many practical soft properties, where violation for a small fraction of time is acceptable, can be phrased as few-flow properties (see Tab. 1).

For example, an operator may be interested in the probability that a customer reaches a certain prefix for SLA compliance (see $\text{Reachable}_{u \to d}$, which is a single-flow property) or that this traffic traverses a monitoring node ($\text{Waypoint}_{u \to d}$). Alternatively, she may want to detect resource allocation deficiencies by checking how frequently a given flow exits at a suboptimal West coast egress ($\text{Egress}_{u \to d}$) or traverses too many hops ($\text{PathLength}_{u \to d}$).

Further, given a small set F including the largest flows in the network together with their mean volumes, she may want to inspect how frequently the load on two intercontinental links is balanced (Balanced_F), or how often the flows are isolated (Isolation_F) or together consume more than 80% of a link's capacity (Congestion_F).

Checking properties NetDice supports any single- or multi-flow property ϕ that can be efficiently checked given the forwarding graphs $G_1, \ldots, G_n$ of the n flows in flows(ϕ) for a concrete failure scenario. In particular, we assume the existence of a function Check($\phi, G_1, \ldots, G_n$) returning true iff the forwarding graphs $G_1, \ldots, G_n$ satisfy ϕ. Note that this function does *not* need to deal with control plane protocols or failures as the forwarding plane will already be computed by NetDice.

Our implementation of Check (see §8) relies on simple variants of depth-first searches to verify the properties in Tab. 1 for given forwarding graphs. By adapting the Check implementation, NetDice can easily be extended to verify custom properties not listed in Tab. 1, e.g. by leveraging existing deterministic verifiers. We note that efficient Check implementations are out of scope of this paper.

Limitations In §5, we discuss how NetDice infers $P(\phi)$ for a property ϕ by effectively pruning the space of failures. NetDice's efficiency thereby depends on the size of flows(ϕ). While the approach presented in this paper can theoretically be used to analyze any single- or multi-flow property, the performance of NetDice degrades as the size of flows(ϕ) becomes large (we show this empirically in §8.4). In particular, analyzing properties involving *all* flows in a network is often intractable. However, few-flow properties allow for very efficient analysis using NetDice (see §8).

1. *Local preference* (prefer highest)
2. *AS path length* (prefer lowest)
3. *Origin* (prefer IGP over EGP over INCOMPLETE)
4. *Multi-exit discriminator* (prefer lowest, compared only for announcements from the same neighbor AS)
5. Prefer external over internal announcements
6. *IGP cost* towards egress router (prefer lowest)
7. BGP *peer id* (prefer lowest, serves as tie break)

Table 2: BGP decision process [13, 33]. Announcements are compared in a lexicographic way, starting with step 1.

3.5 Failures and Problem Statement

We now introduce NetDice's problem statement.

Failure distribution Links (and other network components) fail probabilistically. To model this, we define a vector L of random variables $L_e \in \{0, 1\}$ for each link $e \in E$. We interpret $L_e = 1$ and $L_e = 0$ as link e being up, resp. down. We write $P(L)$ to denote the probability mass function of the joint distribution over link failures. This is, the probability $P(L = l)$ for some vector l can be interpreted as the fraction of time the links are up or down according to l.

The distribution $P(L)$ can capture complex dependencies between link failures, including node and shared risk link group (SRLG) failures (§6).

Property distribution Link failures may change the FIB and potentially also whether a property holds. Let ϕ be the (Bernoulli) random variable with values in $\{0, 1\}$ indicating whether a given single- or multi-flow property does (value 1) or does not hold (0). Note that whether the property holds is fully determined for a given link state L (by the Check function, see §8.4), meaning that $P(\phi \mid L)$ is either 0 or 1. We are interested in computing $P(\phi = 1)$.

Problem statement NetDice addresses the following problem statement: Given (i) a network configuration, (ii) external BGP announcements, (iii) a link failure distribution $P(L)$, and (iv) a single- or multi-flow property ϕ, compute the probability $P(\phi = 1)$ of the property being satisfied.

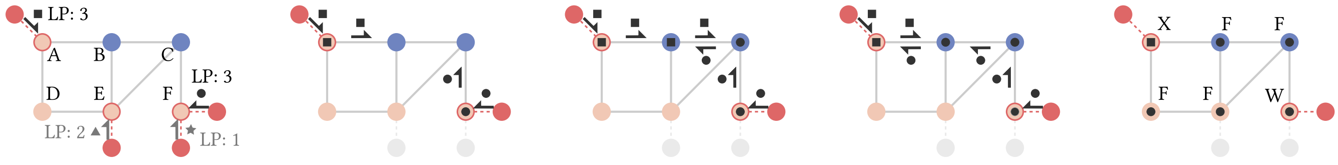
4 MODELLING BGP

We now provide our model of BGP and present an algorithm to simulate BGP for given external announcements.

4.1 Notation and BGP Overview

Peers We denote the set of external nodes, route reflectors and border routers as Ext, Rr and Br, respectively (Ext and V are disjoint). Each node in V and Ext is assigned a unique peer ID. We assume that a BGP session between two nodes is intact as long as the two nodes are connected in the topology.

Announcements BGP peers exchange announcements of paths towards external destinations. In addition to indicating the next hop and the destination prefix, an announcement carries *attributes* including local preference, AS path length, origin, and multi-exit discriminator (MED).



(a) initial state & pre-processing (b) round 1 (c) round 2 (d) round 3 (converged) (e) selected next hops

Figure 4: Simulating BGP using Alg. 1, considering local preference (LP) and IGP costs only. Symbols inside nodes denote the best selected announcements. The BGP session layout is as shown in Fig. 3.

Protocol BGP processes destinations independently, hence our description of the protocol focuses on one destination d. We consider a synchronous model of BGP [33] consisting of multiple rounds, similar to *activation sequences* [3, 13]. This assumes that the converged state of BGP is independent of the message ordering—a practical assumption [13].

First, external peers send announcements to the border routers. Then, in each round until convergence, every node receives announcements from its peers and selects the best announcement according to the BGP decision process described in Tab. 2. This best announcement is sent towards all route reflector clients (except to the peer the announcement was received from), and, if the announcement was received from an external peer, also to all internal peers. The algorithm returns for each node $v \in V$ the next hop field of the best announcement in the converged state. We use $\textsc{Nh}_d(v)$ to denote the selected next hop for destination d at node v.

4.2 Simulating BGP

We now describe how BGP can be simulated.

Pre-processing (Top3) As observed in [13], the first three steps of Tab. 2 induce a global preference relation over announcements. This is, if the graph of BGP sessions is connected, an announcement entering the network will always rule out *all* other announcements that are less preferred according to steps 1–3 of Tab. 2. Hence, we can prune suboptimal announcements in a pre-processing step called *Top3*.

For instance, assume that external peers send announcements as indicated in Fig. 4a. For simplicity, we only consider the local preference (LP) attribute. The announcements ▴ and ⋆ will never be selected as a best announcement in the converged state because ■ and ● have higher local preference. We can hence safely remove ▴ and ⋆ before simulating BGP. Also, we can remove E from the set of border routers as it does not receive any external announcements anymore. Note that failures may break connectivity between groups of BGP routers, which needs to be considered by Top3.

Unfortunately, step 4 of Tab. 2 breaks the global preference relation amongst routes [13]. We can hence not include more steps of Tab. 2 during pre-processing.

Passive nodes Internal nodes only receive announcements from internal peers and have no route reflector clients. Hence, they are passive in the sense that they never *send* any announcement. To determine the converged state of BGP, it suffices to simulate interactions between border routers and route reflectors. Once converged, we can run one BGP round at internal nodes to determine their selected next hops.

Algorithm 1 Simulating BGP in a network partition $X \subseteq V$

```
1: run Top3 (discard suboptimal announcements)
2: while not converged do
3:     for node $n \in \textsc{Top3}(\textsc{Br}, X) \cup (\textsc{Rr} \cap X)$ do
4:         receive announcements from peers
5:         select best announcement according to steps 4–7 of Tab. 2
6:         re-distribute best announcement to peers
7: for each node $x \in X$ do
8:     select next hop $\textsc{Nh}_d(x)$ based on best announcement
```

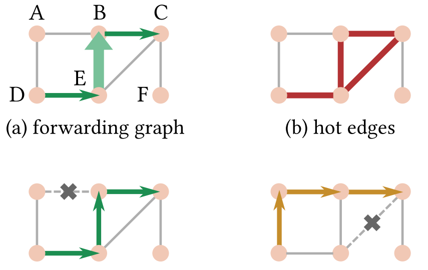
Simulating BGP The two ideas above give rise to the BGP simulation algorithm in Alg. 1. Given a network partition $X \subseteq V$ (it may be $X \neq V$ due to link failures), the algorithm determines the selected next hops for all nodes in X. In Lin. 1, Alg. 1 runs Top3 and discards suboptimal announcements. Lin. 3–6 simulate one BGP round for all non-internal nodes. Here, we write Top3(Br, X) to denote the set of border routers in X remaining after pre-processing. During simulation, only steps 4–7 of Tab. 2 need to be considered.

Example Fig. 4 illustrates an execution of Alg. 1 in the topology from Fig. 3. Announcements ▴ and ⋆ are discarded during Top3 in Lin. 1 (see Fig. 4a). Lin. 2–6 simulate BGP rounds for the 4 nodes A, B, C, and F. In round 1 (see Fig. 4b), nodes A and F receive one announcement each, which is selected (symbols inside nodes). These announcements are forwarded to the respective internal peers. Round 2 is similar (see Fig. 4c). In round 3, both nodes B and C receive ■ and ● (see announcements sent to B and C in round 2, Fig. 4c). At node B, step 6 of Tab. 2 compares the IGP costs 2 and 1 towards the next hops A (for ■) and F (for ●), respectively. This results in the selection of announcement ● at B (see Fig. 4d). Similarly, also C selects ●.

No node changes its decision in any further round and the loop in Lin. 2–6 terminates. Lin. 7–8 select the next hops for all nodes in the network (see Fig. 4e). Nodes A and F select external nodes as next hops, indicating that they forward traffic out of the network.

5 PRUNING THE FAILURE SPACE

We now discuss how NetDice identifies links whose failures are guaranteed not to change whether a property ϕ holds. We first formalize the notions of cold and hot edges (§5.1), and then present how, in addition to shortest path routing, we can incorporate static routes (§5.2) and BGP (§5.3). For simplicity, in §5.2–§5.3 we assume that ϕ is a single-flow property. In §5.4, we discuss how to support multi-flow properties and how to incorporate further protocols.



(a) forwarding graph (b) hot edges (c) failing cold edge (d) failing hot edge

Figure 5: Hot edges for static routes and shortest paths. The link weights are as shown in Fig. 3.

5.1 Cold and Hot Edges

Definition 5.1. Given a network configuration and a flow (u, d), a set of edges $C \subseteq E$ is *cold* iff any combination of link failures in C is guaranteed to not change the forwarding graph for (u, d). An edge is cold iff it belongs to a set of cold edges. An edge is *hot* iff it is not cold.

In the following, we focus on identifying hot edges, and declare their complement as cold. While one could trivially declare *all* edges as hot, this would not allow for pruning any failures. Therefore, our goal is to identify as few hot edges as possible, while still ensuring the complement to be cold.

Shortest paths As we have illustrated in §2, for shortest paths routing, all edges along the[5] shortest path are hot, while all other edges are cold (see Fig. 2). In this case, our choice of hot edges is *optimal*: Any failure of hot edges is guaranteed to change the forwarding graph. However, for more complex protocols (and their combination), we generally cannot guarantee optimality (see §5.2).

5.2 Incorporating Static Routes

We now discuss how to additionally support static routes. Let $\textsc{Static}_d \subseteq V \times V$ be the set of static routes in the network for a destination d. Consider the topology of Fig. 3 and assume that in addition to shortest paths, the administrator has configured $\textsc{Static}_C = \{(E, B)\}$ (see large arrow in Fig. 5a). The flow (D, C) is forwarded as shown in Fig. 5a. D determines the shortest path towards C, which is D–E–C. Hence, it forwards traffic to E. The static route at E deflects traffic from the initial shortest path and forwards it to B instead. At B, traffic again uses the shortest path towards C.

Local stability For a set of edges $C \subseteq E$ to be cold, it is sufficient that every node in the forwarding graph does not change its forwarding behavior for the flow under consideration if any combination of links in C fails. We formalize this observation in Def. 5.2 and Lemma 5.3.

Definition 5.2. A node is *locally stable* under a set of link failures w.r.t. a flow if packets of that flow are forwarded identically by the node both with and without the failures.

Algorithm 2 Hot edges for static routes and shortest paths

```
1: procedure HotSpStatic($u$, $d$, $E_{\text{fwd}}$, $L$)
2:     $\mathcal{D} \leftarrow \{u\} \cup \{y \mid (x, y) \in \textsc{Static}_d \cap E_{\text{fwd}}\}$      ▹ decision points
3:     $\mathcal{H} \leftarrow \textsc{AllSp}(\mathcal{D}, \{d\}, L)$      ▹ all shortest paths $\mathcal{D} \to d$
4:     return $\mathcal{H} \cup (\textsc{Static}_d \cap E_{\text{fwd}})$      ▹ traversed static routes
5: procedure AllSp($\mathcal{S}$, $\mathcal{T}$, $L$)      ▹ all shortest paths $\mathcal{S} \to \mathcal{T}$
6:     return $\bigcup_{s \in \mathcal{S}, t \in \mathcal{T}} \textsc{Sp}_L(s, t)$      ▹ shortest path $s \to t$
```

Lemma 5.3. *Let (u, d) be a flow with forwarding graph f, and $C \subseteq E$ a set of edges. Further, assume that every node in f is locally stable w.r.t. flow (u, d) under any subset of C failing. Then, C is cold.*

Lemma 5.4 (see below) states that under shortest paths routing, local stability is transitive. Hence, to apply Lemma 5.3 we only need to prove local stability for few nodes, namely (i) the source node of the flow, (ii) all start and (iii) all end nodes of traversed static routes. All other nodes on the forwarding graph will be locally stable by Lemma 5.4. Note that (iii) is required as static routes can deflect the forwarding graph from the initial shortest path, as shown in Fig. 5a.

Lemma 5.4. *Let $x, y \in V$ be nodes using shortest path routing for a flow (u, d), where y is the next router along the shortest path from x to d. If x is locally stable under some failures w.r.t. flow (u, d), then so is y.*

Proof. If the failures would change the shortest path p_y from y to d, also the shortest path p_x from x to d would change, because p_y is necessarily a subpath of p_x. □

Example To ensure local stability of the ingress D, links D–E and E–C of the shortest path D–E–C must be hot. Note that this is the case even though E–C does not lie on the forwarding graph: if this link fails (see Fig. 5d), the shortest path from D to C changes to D–A–B–C. To ensure (ii), we mark link E–B as hot (the source of a static route is locally stable as long as the output link does not fail). For (iii), we need to mark the link B–C as hot. This makes sure the shortest path between B and C does not change under cold edge failures.

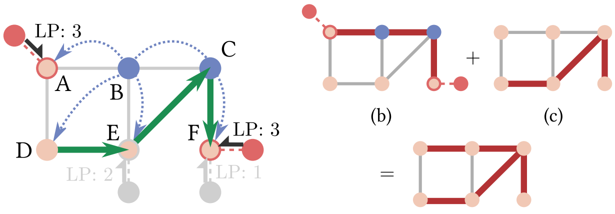
Fig. 5b summarizes the hot edges for this example. Any failures of other edges will not change the forwarding graph. Fig. 5c shows an example of such a cold edge failure.

General algorithm Alg. 2 determines hot edges in an arbitrary network for link state L, ensuring local stability for (i—iii) above. The procedure HotSpStatic takes as input a flow $(u, d) \in V \times V$, the edges E_{fwd} of the forwarding graph for that flow, and the link state L. First, in Lin. 2 it collects u and all end points of traversed static routes in a set $\mathcal{D}$. These nodes are called *decision points*. Next, in Lin. 3 it determines all edges along a shortest path between any $r \in \mathcal{D}$ and d. Here, $\textsc{Sp}_L(s, t)$ is the set of all edges along the shortest path from s to t under link states L. Finally, in Lin. 4 the procedure adds links used by any encountered static route to $\mathcal{H}$. For our example, Alg. 2 returns the red edges in Fig. 5b.

Correctness Lemma 5.5 shows that Alg. 2 is correct.

Lemma 5.5. *The complement $C = E \setminus \mathcal{H}$ of the set $\mathcal{H}$ returned by* HotSpStatic *(Alg. 2) is cold.*

[5] When referring to *the* shortest path, we refer to the unique path chosen by the routing protocol (e.g., OSPF), even though there may be multiple shortest paths. In §5.4, we discuss how to incorporate ECMP.



(a) forwarding graph (b) (c) (d) final hot edges

Figure 6: Hot edges for BGP combined with shortest paths.

Proof. Assume any subset of C fails. According to Lemma 5.3, we only need to prove local stability for nodes in the forwarding graph f. Due to Lin. 3, all decision points are locally stable. Due to Lin. 4, all nodes in f with a configured static route are locally stable. Finally, using a simple inductive argument leveraging Lemma 5.4, it can be shown that every other node in f is also locally stable. □

Unlike with shortest paths only, the set $\mathcal{H}$ is generally an over-approximation of hot edges: while any failure of a hot edge not carrying any traffic *may* change the forwarding graph (e.g., see Fig. 5d), this is not necessarily the case.

5.3 Incorporating BGP

Next, we discuss how to identify hot edges for BGP. Consider again the example in Fig. 4, where next hops are selected for an external destination d. Together with shortest paths routing, the flow (D, d) is forwarded as shown in Fig. 6a.

At a high level, we have to make sure that any cold edge failures do not change, at each node in the forwarding graph, (i) the selected BGP next hop r and (ii) the next node towards r according to the internal routing protocol. We have discussed the main ideas for (ii) in §5.2. Next, we inspect (i).

Guarding the selected next hop There are multiple reasons why link failures may affect the selected next hop of a node n. First, failures may disconnect n from a BGP peer, leading to n not receiving that peer's announcements anymore. In our example, failing links A–B and A–D would prevent B from receiving announcement ■ sent by A in round 1 (see Fig. 4b). Second, the IGP costs involved in step 6 of Tab. 2 when comparing announcements can change due to failures. For instance, in round 3 of our example (Fig. 4d), node B selects ● over ■ due to lower IGP cost. If link B–C fails, the IGP cost from B to F changes to 4, making B select ■ instead. Third, failures may change announcements sent by a peer of n and thereby transitively influence the decision of n.

Combining these thoughts with our insights from §5.2 gives rise to Alg. 3, which collects hot edges for the BGP protocol. Procedure HotBgp takes as input a flow (u, d), where $u \in V$ is a node and d is an external destination, the edges E_{fwd} of the forwarding graph for that flow, and link states L. The algorithm performs three main steps.

Step 1: Ensuring local stability for route reflectors First, in Lin. 2–4 we perform BGP pre-processing and determine the set of border routers and route reflectors in the partition X. Then, in Lin. 5 we mark the shortest paths from any route reflector to any border router as hot. This ensures that in X, (i) all route reflectors remain connected with all border routers, and (ii) the IGP costs compared in step 6 of Tab. 2 during *any* round of Alg. 1 do not change. In our example, this step marks links A–B, B–C, and C–F as hot, see Fig. 6b.

Algorithm 3 Hot edges for BGP

```
 1: procedure HotBgp($u$, $d$, $E_{\text{fwd}}$, $L$)
 2:   $X \leftarrow$ nodes in the same partition as $u$ under $L$
 3:   $\text{Br}_L \leftarrow \text{Top3}(\text{Br}, X)$                ▷ BGP pre-processing (§4.2)
 4:   $\text{Rr}_L \leftarrow \text{Rr} \cap X$
 5:   $\mathcal{H} \leftarrow \text{AllSp}(\text{Rr}_L, \text{Br}_L, L)$          ▷ all shortest paths (Alg. 2)
 6:   $\mathcal{D} \leftarrow \{u\}$                                   ▷ decision points
 7:           $\cup \{y \mid (x, y) \in \text{Static}_d \cap E_{\text{fwd}}\}$
 8:           $\cup \{y \mid (x, y) \in E_{\text{fwd}} \wedge \text{Nh}_d(x) \neq \text{Nh}_d(y)\}$
 9:   for each $x \in \mathcal{D}$ do
10:     $\mathcal{H} \leftarrow \mathcal{H} \cup \text{Sp}_L(x, \text{Nh}_d(x))$      ▷ shortest path $x \rightarrow \text{Nh}_d(x)$
11:   $\mathcal{H} \leftarrow \mathcal{H} \cup (\text{Static}_d \cap E_{\text{fwd}})$        ▷ traversed static routes
12:   if $\text{Rr}_L = \emptyset$ then
13:     $\mathcal{H} \leftarrow \mathcal{H} \cup \text{AllSp}(\{u\}, \text{Br}_L)$           ▷ ensure connectivity
14:   return $\mathcal{H}$
```

Step 2: Ensuring local stability for decision points Similarly as in Alg. 2, Lin. 6–10, ensure that cold edge failures can not change the forwarding behavior of a set $\mathcal{D}$ of decision points. $\mathcal{D}$ includes the same nodes as in Alg. 2, but additionally includes nodes where the next hop changes (see Lin. 8). This may for example happen if adjacent nodes are clients of different route reflectors sending different announcements. In Lin. 9–10, we mark all edges on any shortest path from a decision point to the selected next hop as hot. Together with step 1, step 2 ensures local stability of decision points under shortest paths routing. Like in Alg. 2, in Lin. 11 of Alg. 3 we mark all links used by a static route in E_{fwd} as hot in order to ensure local stability of nodes with static routes.

In our example, the set of decision points is $\mathcal{D} = \{\text{D}, \text{F}\}$ (there are no static routes). In Lin. 6–11, the algorithm marks links D–E, E–C, and C–F as hot (see Fig. 6c).

Step 3: Ensuring connectivity If there is at least one route reflector in X, Lin. 5–11 ensure that all route reflectors, border routers and nodes on the forwarding graph remain connected. Hence, all BGP sessions between these nodes remain intact under cold edge failures. However, Lin. 5 does not enforce border routers to be connected if $\text{Rr}_L = \emptyset$. As a technical detail, we have to separately ensure connectivity in this case by connecting u with all border routers in Lin. 13.

In our example, $\text{Rr}_L \neq \emptyset$. Hence, Lin. 12–13 do not add any hot edges. The final set of hot edges is shown in Fig. 6d.

Correctness Lemma 5.6 shows that Alg. 3 is correct.

Lemma 5.6. *The complement* $C = E \setminus \mathcal{H}$ *of the set* $\mathcal{H}$ *returned by* HotBgp *(Alg. 3) is cold.*

Proof Sketch. We provide a full proof in App. A.2. First, we show stability of BGP decisions by arguing that in every iteration of Alg. 1, all sent announcements do not depend on failures in C. Then, we prove local stability for all decision points and conclude by levering Lemma 5.4 and Lemma 5.3. □

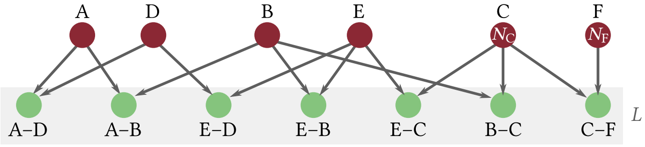


Figure 7: Modelling node failures as a Bayesian Network.

5.4 Extensions

ECMP With ECMP, traffic is forwarded along multiple shortest paths, not all of which would be marked as hot by Alg. 2 and Alg. 3. We can easily incorporate ECMP by additionally marking all edges in E_{fwd} as hot.

Multi-flow properties Extending our approach to multi-flow properties is straightforward. In particular, given a multi-flow property ϕ with flows(ϕ) = $\{(u_1, d_1), \dots, (u_n, d_n)\}$, we can just union the hot edges for the individual flows to obtain the hot edges for ϕ. This is, for E^i_{fwd} being the forwarding graph of flow (u_i, d_i), we compute $\mathcal{H} = \bigcup_{i=1}^{n} \textsc{HotBgp}(u_i, d_i, E^i_{\text{fwd}}, L)$. Any failure of a cold edge $e \notin \mathcal{H}$ is guaranteed to not change the forwarding graph for any flow in flows(ϕ).

Further protocols The general concept of cold edges can in principle be applied to any routing protocol. While identifying cold edges for a new protocol requires manual effort and domain-specific insights, we believe that many of the ideas developed here (e.g., local stability) can be re-used.

6 PROBABILISTIC FAILURE MODEL

We now describe how to represent the failure distribution $P(L)$. In NetDice, failure models are expressed as dependencies amongst link failures. For example, a node or shared risk link group (SRLG) failure can be modeled as a simultaneous failure of multiple links. Similar to [36, 37], we model such dependencies using a Bayesian Network (BN) representing the distribution $P(L)$. Each variable X in the BN is binary (values 0 or 1), where $X = 0$ indicates a failure. Each link $e \in E$ is associated with a variable L_e in the BN, and more variables and dependencies can be added. Because any discrete probability distribution with arbitrary dependencies can be represented by a BN, this approach is very general.

Example: Independent link failures A BN modelling independent link failures just consists of one variable L_e per link $e \in E$. The user of NetDice simply specifies for each link e the failure probability $P(L_e = 0)$. The distribution represented by the BN is: $P(L) = \prod_{e \in E} P(L_e)$.

Example: Node and SRLG failures Node and SRLG failures can be modeled by adding a variable N_v for each node $v \in V$ (resp. SRLG) to the BN and connecting them with the variables for adjacent links (resp. links belonging to the SRLG).

For example, the BN modelling node failures for the topology in Fig. 3 is shown in Fig. 7. Assuming the probability of a node (resp. link) failure is 0.1 (resp. 0.01), the BN would declare:

$$P(N_C = 0) = 0.1 \qquad P(L_{C\text{–}F} = 0 \mid N_C = 0 \lor N_F = 0) = 1$$
$$P(N_F = 0) = 0.1 \qquad P(L_{C\text{–}F} = 0 \mid N_C \neq 0 \land N_F \neq 0) = 0.01.$$

Such probabilities can be estimated, e.g., from network log data [6, 20, 38]. Note the dependency of $L_{C\text{–}F}$ on its parents N_C and N_F: the link fails if any of its adjacent nodes fails.

By adding more variables to the BN with similar dependencies, one can express more complex failure models such as correlated SRLG failures.

Inference We will see in §7 how NetDice repeatedly queries the BN model for probabilities $P(L)$ or any of its marginals when computing $P(\phi)$. NetDice uses Variable Elimination [41] to compute exact marginals from the failure model BN.

7 EXPLORING THE FAILURE SPACE

We now show how NetDice explores the space of link failures to compute $P(\phi)$ for a single- or multi-flow property ϕ.

The target probability $P(\phi)$ can be expanded as follows:

$$P(\phi) \;=\; \textstyle\sum_L P(\phi \mid L) \cdot P(L). \tag{1}$$

We have seen in §6 how to compute $P(L)$ for a given link state L. Further, given L the value of ϕ is deterministic (i.e., $P(\phi \mid L)$ is either 0 or 1): We can construct the forwarding graphs for flows(ϕ) and verify the property using the Check function (see §3.4). Hence, we can rewrite (1) to:

$$P(\phi) \;=\; \textstyle\sum_{L \text{ s.t. } \phi \text{ holds for } L} P(L). \tag{2}$$

Skipping cold edge failures Rather than exploring *all* possible states L to compute (2), we can leverage the idea of cold edges from §5 to visit only a few states. The main idea of NetDice's exploration algorithm is to recursively introduce failures of hot edges while skipping failures of cold edges. Due to the result in Lemma 7.1, the latter can be covered implicitly.

Lemma 7.1. *Let $C \subseteq E$ be cold for a link state L and every flow in flows(ϕ) for a single- or multi-flow property ϕ. Further, let L' be the link state L where some links in C fail. Then, $P(\phi \mid L) = P(\phi \mid L')$.*

Proof. By Def. 5.1, any failures of links in C will not change the forwarding graphs for any flow in flows(ϕ). □

States We define a *state* to be a function $s\colon E \longrightarrow \{1, 0, ?\}$ assigning to each link one of 1 (up), 0 (down) or "?" (undecided). The value 1 "locks" a link (see e.g. D–E in L_3 of Fig. 2), while "?" means that the link is up, but may fail later. A state should not be confused with the random vector L. A state is *ground* if no link is assigned "?". A state s' is *compatible* with s iff $s(e) \in \{0, 1\} \implies s'(e) = s(e)$ and *compat*(s) is the set of ground states compatible with s. The state *fill(s)* assigns 1 to all undecided links in s and $s[e \longleftarrow c]$ (resp. $s[\mathcal{E} \longleftarrow s]$) is the state s where link e (resp. every link in the set $\mathcal{E}$) is assigned $c \in \{0, 1\}$. We define *prob*(s) to be the marginal probability of L over the ground states compatible with s:

$$\text{prob}(s) := P\left(\textstyle\bigwedge_{e \in E \text{ s.t. } s(e) \neq ?} L_e = s(e)\right). \tag{3}$$

This quantity can be computed using marginal inference in the Bayesian Network representing $P(L)$ (see §6).

Algorithm 4 Failure exploration to compute $P(\phi)$

```
1:  procedure EXPLORE($s$)
2:    $L \leftarrow \text{fill}(s),\ \mathcal{H} \leftarrow \emptyset,\ \rho \leftarrow 0$
3:    for each flow $(u_i, d_i) \in \text{flows}(\phi)$ do
4:      compute forwarding graph $(V^i_{\text{fwd}}, E^i_{\text{fwd}})$ for $(u_i, d_i)$ under $L$
5:      $\mathcal{H} \leftarrow \mathcal{H} \cup \text{HOTBGP}(u_i, d_i, E^i_{\text{fwd}}, L)$        ▷ Alg. 3
6:    $\mathcal{H}^? \leftarrow \mathcal{H} \cap \{e \mid e \in E \wedge s(e) = ?\}$        ▷ undecided hot edges
7:    if CHECK($\phi, (V^1_{\text{fwd}}, E^1_{\text{fwd}}), \ldots, (V^n_{\text{fwd}}, E^n_{\text{fwd}})$) then        ▷ §3.4
8:      $\rho \leftarrow \rho + \text{prob}(s[\mathcal{H}^? \leftarrow 1])$        ▷ marginal prob. (§6)
9:    $s' \leftarrow s$
10:   for $l \in \mathcal{H}^?$ do
11:     $s' \leftarrow s'[l \leftarrow 0]$        ▷ introduce failure
12:     $\rho \leftarrow \rho + \text{EXPLORE}(s')$        ▷ explore recursively
13:     $s' \leftarrow s'[l \leftarrow 1]$        ▷ "lock" link
14:   return $\rho$        ▷ $\rho = \sum_{L \in \text{compat}(s)} P(\phi \mid L) \cdot P(L)$
```

Failure exploration NetDice explores a tree of states, starting with the all-undecided state s_0, to build up the sum in (2). When visiting a state, it checks whether ϕ holds, assuming all undecided links are up. It then introduces failures of undecided hot edges and recursively visits these new states.

This procedure is detailed in Alg. 4. The function EXPLORE is initially invoked with $s = s_0$ and begins by setting all undecided links to "up" (Lin. 2). In Lin. 3–5, we construct all forwarding graphs relevant for ϕ and collect hot edges as presented in §5. Next, Lin. 6 finds all undecided hot edges. Then, we check the property ϕ for the forwarding graphs (Lin. 7). If the property holds (Lin. 8), we compute the marginal probability of link states differing from L at most by failed undecided cold edges (see §6) and add it to ρ. After Lin. 8, the value of ρ is $\sum_{L \in \text{compat}(s[\mathcal{H}^? \leftarrow 1])} P(\phi \mid L) \cdot P(L)$ due to Lemma 7.1. Finally, in Lin. 9–13 we explore all states with at least one additional failure of an undecided hot edge.

Lemma 7.2 is proven in App. A.2. Because compat(s_0) covers all link states, it follows that Alg. 4 correctly computes (1).

LEMMA 7.2. *EXPLORE(s) returns* $\sum_{L \in compat(s)} P(\phi \mid L) \cdot P(L)$.

In the worst case (if for every link state, all undecided links are hot), Alg. 4 visits $2^{|E|}$ states. However, our experiments show that much fewer states are visited in practice (see §8).

Bounded exploration Alg. 4 can be stopped early, in which case its output is a lower bound p_l of $P(\phi)$. By tracking how much probability mass of $P(L)$ has been covered and leveraging the fact that $P(\phi \mid L)$ is at most 1, NetDice also obtains an upper bound p_h and returns to the user the interval $[p_\text{l}, p_\text{h}]$, which is *guaranteed* to contain $P(\phi)$. Such a result is often sufficient in practice, given the interval is small. For example, if NetDice determines that the probability of reachability is in $[0.99991, 0.99998]$, a reachability of four 9's is guaranteed.

As presented, Alg. 4 performs a depth-first search. NetDice actually uses a breadth-first variant, where states are roughly explored according to their probability (i.e., most likely states first). As a result, the *imprecision* $p_\text{h} - p_\text{l}$ drops rapidly and the tool can stop exploration as soon as a desired target imprecision is reached. As we show in §8, this approach allows trading little precision for large speedups.

8 IMPLEMENTATION AND EVALUATION

We now evaluate our approach using an end-to-end implementation of NetDice consisting of ≈3k lines of Python code. [6] As input, NetDice accepts a network configuration, eBGP announcements after import policies, a failure model, and a property. NetDice currently supports link and node failure models, and the properties in Tab. 1. The tool can easily be extended to other failure models and additional single- and multi-flow properties.

After introducing our methodology (§8.1), we analyze NetDice's performance for different real-world topologies with synthetic configurations (§8.2–§8.3). Next, we discuss how increasing the number of flows for multi-flow properties impacts the performance of NetDice (§8.4). Finally, we demonstrate NetDice's success in analyzing the real-world configurations of a nation-level ISP (§8.5).

Overall, we show that NetDice verifies few-flow properties for networks with hundreds of links in few minutes with imprecision below 10^{-4}, sufficient for four 9s availability guarantees.

8.1 Methodology and Dataset

Our experiments run on a machine with 32 GB of RAM and 12 cores at 3.7 GHz. The presented numbers are for sequential execution (verification of a property is not parallelized), however one can easily run multiple queries in parallel.

For the synthetic experiments (§8.2–§8.4), we combine 80 topologies from the Topology Zoo [27] (topologies with ≥ 50 links) with the 10 publicly available[7] tier 1 and transit topologies from [30]. The resulting dataset $\mathcal{T}$ includes 90 topologies containing between 18 (resp. 50) and 754 (resp. 2 320) nodes (resp. links). We assume uniform link weights 1 and do not setup static routes.

In all our experiments, we use a node failure model with failure probabilities 0.0001 (for nodes) and 0.001 (for links). This is in line with previous studies [20, 38] which report WAN links being up between 99.9 to 99.99 percent of time.

8.2 Different Network Sizes

We now analyze NetDice's performance for verifying a waypoint property in the topologies $\mathcal{T}$. The actual property check (call to CHECK in Lin. 7 of Alg. 4) only accounts for a negligible part of the runtime (all properties in Tab. 2 are decided by simple variants of depth-first searches). Hence, the runtime is only noticeably affected by the number of flows involved in the property. The following results are thus representative for *any* single-flow property with efficient CHECK function (see §8.4 for multi-flow properties).

Because the topologies in $\mathcal{T}$ do not include any BGP configurations, we create synthetic BGP setups with 2 route reflectors and 10 border routers having 2 external peers each. This is in line with previous studies [9] on next hop diversity in real networks (see §8.3 for other setups). Following the best common practices for ISPs [21], the route reflectors (resp. border routers) are randomly sampled with a bias towards (resp. away from) the network center. [8]

[6] Our implementation is available on GitHub: https://github.com/nsg-ethz/netdice

[7] https://inl.info.ucl.ac.be/content/mrinfo (Accessed: 04.02.2020)

[8] The sampling probability decreases (resp. increases) according to the squared distance to the center in terms of number of links.

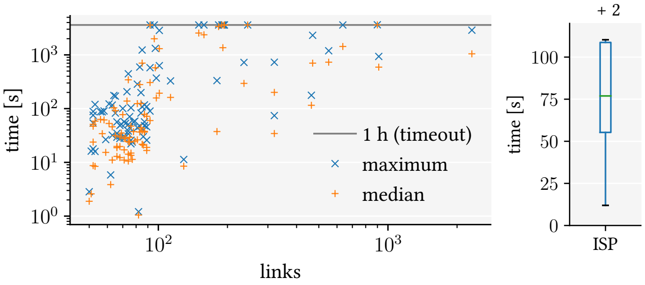


Figure 8: Runtime for verifying a single-flow waypoint property in different topologies by size (left) and for the real ISP configuration (right; outliers 233 s and 497 s not shown).

For the external announcements, we use the *worst case* scenario where all external peers send the *same attributes*. As a result, Top3 (see §4.2) does not remove any announcements and the number of hot edges added in Lin. 5 and Lin. 13 of Alg. 3 is *maximal*. Note that NetDice's runtime is primarily determined by the number of hot edges, which depends on the size of Br_L (see Alg. 3, Lin. 3) but *not on the specific announcement attributes* (the runtime of Top3 is negligible). The presented results are hence representative for *any* scenario with additional, less preferred announcements.

Runtime We measure the runtime of NetDice verifying a random waypoint property of a single random flow for a target imprecision of 10^{-4} (i.e., we stop exploration once the target is reached). For each network in $\mathcal{T}$, we run 10 experiments with fresh random choices. Fig. 8, left shows for each network the maximum and median inference time. While inference in larger networks is generally more expensive, most networks (including the largest network, 2 320 links) can be analyzed within 1 h. Only for 12 (resp. 6) networks, at least one experiment times out after 1 h (resp. 3 h). For 7 out of these, the imprecision at the point of timeout is below 10^{-3}.

We note that the runtime is currently dominated by Variable Elimination (VE) for Bayesian network inference (see §6). Experiments showed that with link failures only (where VE is not required), performance improves by a factor 5–10×. While the expressiveness of our failure model currently comes at a cost, the performance can likely be significantly improved by leveraging more advanced inference algorithms.

To summarize, NetDice can efficiently and precisely analyze most networks with hundreds of links in few minutes.

Fraction of hot edges We also collect the fraction of edges marked as hot in the first 10 visited states of the previous experiments and plot the CDF in Fig. 9, left. For 50 (resp. 80) percent of the states, less than 32 (resp. 42) percent of edges are hot. This clearly shows that NetDice effectively prunes the failure space already in the first few states.

Evolution of imprecision Fig. 9, right shows how NetDice reduces the imprecision while exploring states for the Colt network (191 links). Observe that an imprecision of 10^{-3} is already reached within 100 to 1 000 states (i.e., few seconds).

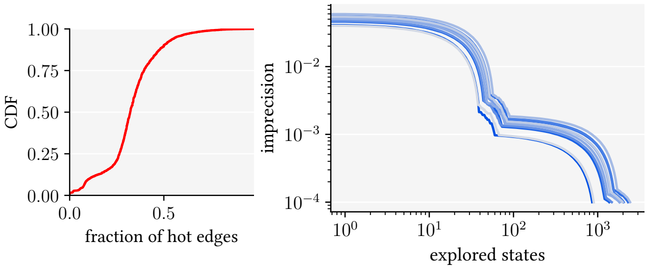


Figure 9: Left: Fraction of hot edges in the first 10 visited states for the experiments of Fig. 8, left. Right: Precision evolution (10 runs) for Colt.

In Fig. 1, we compare the imprecision traces of NetDice and partial exploration for the Colt network and a link failure model. [9] To reach an imprecision of 10^{-4}, partial exploration needs to consider *almost 600 times more states* than NetDice (1 107 359 vs. 1 854 states for NetDice).

8.3 Different BGP Setups

We now inspect the performance of NetDice under an increasing number of route reflectors and border routers. Note that by increasing these numbers, we introduce more dependencies in the BGP protocol and hence increase the number of hot edges (see Lin. 5 and Lin. 13 in Alg. 3).

We re-run the experiments from §8.2 for the Uninett2010 topology (74 nodes, 101 links), but vary the number of border routers (see Fig. 10a) and route reflectors (see Fig. 10b). Note that in practical networks, only few prefixes see more than 20 equally preferred next-hops [9]. As expected, both the median runtime and its variance generally increase with more border routers and route reflectors. Still, the network can be analyzed within 1 hour for all setups.

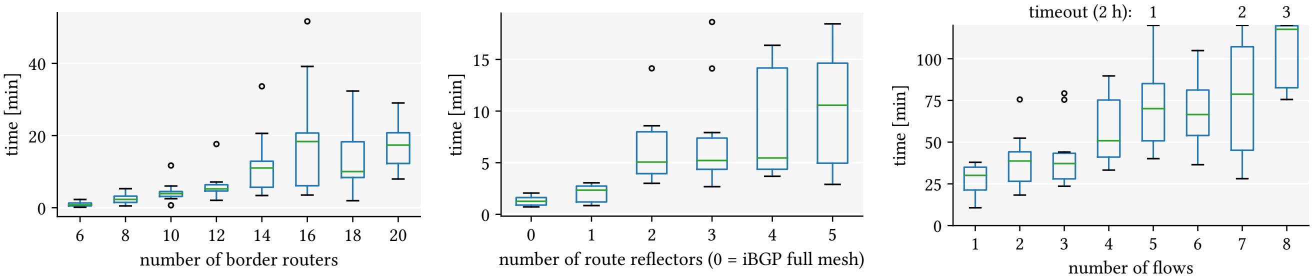
8.4 Multi-Flow Properties

Next, we inspect the performance of NetDice for multi-flow properties. Recall that a flow corresponds to a pair of ingress router and destination prefix, and does *not* refer to a UDP/TCP flow. In particular, we let NetDice verify a congestion property (see Tab. 1) for $k = 1, \ldots, 8$ random flows with distinct destinations and random flow volumes in the AS 3549 topology (235 links, 10 runs, target imprecision 10^{-4}). Instead of randomly sampling border routers and external peers, we use the 21 external peers known for the topology. [10] Route reflectors and announcements are chosen as in §8.2.

We present the results in Fig. 10c. As expected, more flows lead to slower inference (due to more hot edges, see Lin. 3–5 in Alg. 4). While NetDice cannot scalably analyze congestion involving *all* flows in a network, it is a useful and efficient tool for analyzing the interaction of the few largest flows, which often already cover a significant amount of traffic.

[9] With node failures, the imprecision for partial exploration can not be computed in closed form and the optimal order of visiting states is not known a priori. Hence, our comparison uses a link failure model.

[10] Sampling 10 new border routers for each destination (as done in §8.2) would not be realistic. Most networks have only few border routers.



(a) More border routers.

(b) More route reflectors.

(c) More flows for a congestion property.

Figure 10: Varying the number of (a) border routers and (b) route reflectors in the Uninett2010 network for a single-flow waypoint property. (c) Increasing the number of flows for a congestion property in the AS 3549 network.

8.5 Real-Wold Configuration

We finally extract the IGP and BGP settings from the raw configuration files of a nation-level ISP comprising around 100 nodes and 180 links. [11] Like in §8.2, we verify a random single-flow waypoint property (10 runs, imprecision 10^{-4}) under worst-case announcements (we argue why doing so is representative in §8.2). The runtimes are summarized in Fig. 8, right. The median runtime is less than 2 min. We conclude that NetDice can efficiently verify properties for real-world network configurations.

9 RELATED WORK

Control plane analysis Our work is complementary to a large amount of recent work on control plane analysis. Batfish [16] and C-BGP [32] take as input a configuration and a concrete environment (including failures), simulate the control plane, and analyze the resulting data-plane. This however takes non-negligible time, preventing the effective analysis of many environments. Similarly to NetDice, the latest generation of network analyzers such as Minesweeper [4], BagPipe [40], ARC [19], Tiramisu [1], and ERA [12], are able to verify properties considering a wide range of failure scenarios. While useful, none of these approaches reasons about the probability that a network is in a given state, which precludes inference on the probability of a property holding.

Data-plane analysis Many prior works focus on data-plane analysis with systems such as Anteater [29], HSA [25], NetPlumber [24], and VeriFlow [26]. While useful, the results of data-plane analysis are limited to the considered failure scenario. These tools cannot be used for pro-active verification and probabilistic reasoning, unlike NetDice.

Probabilistic network languages ProbNetKat [17] brings probabilistic extensions to NetKat [2, 18]. The authors of [35] give a foundation for building solvers for ProbNetKat models. While ProbNetKat [17] can model the data-plane, it cannot capture control plane protocols as it does not model state at routing nodes. NetDice supports these protocols with custom inference procedures that scale to real-world networks.

Traffic engineering under probabilistic failures A complementary line of work [6, 8] studies synthesis of forwarding rules for traffic engineering in the face of probabilistic failures.

Lancet [8] finds a link-based protection routing that is congestion-free with high probability. Like NetDice, it recursively explores a tree of failure scenarios. In contrast to NetDice, which uses custom algorithms to determine equivalent scenarios and prune the search space at *each* visited state, Lancet's Divide-and-Conquer approach leverages linear programming to decide if a visited state represents a single set of equivalent scenarios (which can be pruned) or has to be divided and analyzed recursively.

TEAVAR [6] optimizes bandwidth allocation subject to a target availability probability in tunnel-based WAN routing, leveraging ideas from financial risk theory. Unlike NetDice, TEAVAR explores failures according to a fixed order and does not determine or prune equivalent failure scenarios.

Similarly to NetDice, these works support complex correlated failure models and avoid exploring very unlikely scenarios using a "cutoff". Unlike NetDice, which supports BGP and shortest-paths routing, they target path-based WAN routing. Addressing powerful but specific synthesis problems, they complement NetDice's verification of more diverse but simpler properties.

10 CONCLUSION

We presented NetDice, a scalable and precise tool for probabilistic network verification. NetDice is based on a novel inference algorithm able to effectively prune the failure space for BGP and common IGPs. We implemented NetDice and evaluated it on real-world configurations. NetDice can verify relevant properties for networks with hundreds of links within minutes with high precision.

NetDice's notion of cold edges may also prove useful in non-probabilistic settings (e.g. to speed up existing verifiers). We encourage future work to explore this direction.

Ethical issues This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We thank our shepherd Sanjay Rao and the anonymous reviewers for their helpful feedback. This work was partially supported by an ETH Research Grant ETH-03 19-2.

[11] Unfortunately, Batfish [16] could not parse the configurations directly, forcing us to extract the information ourselves.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks *(POPL '14)*.

[3] Anindya Basu, Chih-Hao Luke Ong, April Rasala, F. Bruce Shepherd, and Gordon Wilfong. 2002. Route Oscillations in I-BGP with Route Reflection. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pittsburgh, Pennsylvania, USA) *(SIGCOMM '02)*. ACM, New York, NY, USA, 235–247. https://doi.org/10.1145/633025.633048

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 155–168.

[5] Christopher M. Bishop. 2006. *Pattern recognition and machine learning*. Springer, New York.

[6] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. 2019. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, Beijing China, 29–43. https://doi.org/10.1145/3341302.3342069

[7] Lawrence D. Brown, T. Tony Cai, and Anirban DasGupta. 2001. Interval Estimation for a Binomial Proportion. *Statist. Sci.* 16, 2 (05 2001), 101–133. https://doi.org/10.1214/ss/1009213286

[8] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. 2019. Lancet: Better Network Resilience by Designing for Pruned Failure Sets. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (Dec. 2019), 1–26. https://doi.org/10.1145/3366697

[9] Jaeyoung Choi, Jong Han Park, Pei chun Cheng, Dorian Kim, and Lixia Zhang. 2011. Understanding BGP next-hop diversity. In *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 846–851. https://doi.org/10.1109/INFCOMW.2011.5928930

[10] Luca Cittadini, Stefano Vissicchio, and Giuseppe Di Battista. 2010. Doing don'ts: Modifying BGP attributes within an autonomous system. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 293–300.

[11] Mary Kathryn Cowles and Bradley P. Carlin. 1996. Markov Chain Monte Carlo Convergence Diagnostics: A Comparative Review. *J. Amer. Statist. Assoc.* 91, 434 (1996), 883–904. https://doi.org/10.1080/01621459.1996.10476956

[12] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)* (Savannah, GA, USA). USENIX Association, USA, 217–232.

[13] N. Feamster and J. Rexford. 2007. Network-Wide Prediction of BGP Routes. *IEEE/ACM Transactions on Networking* 15, 2 (April 2007), 253–266. https://doi.org/10.1109/TNET.2007.892876

[14] Ashley Flavel, Jeremy McMahon, Aman Shaikh, Matthew Roughan, and Nigel Bean. 2010. BGP route prediction within ISPs. *Computer Communications* 33, 10 (2010), 1180–1190.

[15] Ashley Flavel, Matthew Roughan, Nigel Bean, and Aman Shaikh. 2008. Where's Waldo? practical searches for stability in iBGP. In *IEEE International Conference on Network Protocols. ICNP 2008*. IEEE, 308–317.

[16] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. USENIX Association, Oakland, CA, 469–483.

[17] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems (ESOP '16)*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–309.

[18] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A coalgebraic decision procedure for NetKAT. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 343–355.

[19] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[20] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/2018436.2018477

[21] Barry Raveendran Greene and Philip Smith. 2002. *Cisco ISP essentials*. Cisco Press.

[22] Timothy G Griffin and Gordon Wilfong. 2002. On the correctness of IBGP configuration. In *ACM SIGCOMM Computer Communication Review*, Vol. 32. ACM, 17–29.

[23] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

[24] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX, Lombard, IL, 99–111.

[25] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. USENIX, San Jose, CA, 113–126.

[26] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX, Lombard, IL, 15–27.

[27] Simon Knight, Hung X Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[28] Pierre Simon Laplace. 1812. *Théorie analytique des probabilités*. Ve. Courcier.

[29] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/2018436.2018470

[30] Pascal Mérindol, Virginie Van den Schrieck, Benoit Donnet, Olivier Bonaventure, and Jean-Jacques Pansiot. 2009. Quantifying Ases Multiconnectivity Using Multicast Information. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement (IMC '09)*. Association for Computing Machinery, New York, NY, USA, 370–376.

[31] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. USENIX Association, Santa Clara, CA, 953–967. https://www.usenix.org/conference/nsdi20/presentation/prabhu

[32] Bruno Quoitin and Steve Uhlig. 2005. Modeling the routing of an autonomous system with C-BGP. *IEEE network* 19, 6 (2005), 12–19.

[33] Y. Rekhter, T. Li, and S. Hares. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard). http://www.ietf.org/rfc/rfc4271.txt

[34] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. 2002. BGP routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 197–202.

[35] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. ACM, New York, NY, USA, 557–571. https://doi.org/10.1145/3009837.3009843

[36] M. Steinder and A. S. Sethi. 2002. End-to-end service failure diagnosis using belief networks. In *Network Operations and Management Symposium (NOMS '02)*. 375–390.

[37] M. Steinder and A. S. Sethi. 2002. Increasing robustness of fault localization through analysis of lost, spurious, and positive symptoms. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1. 322–331 vol.1.

[38] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. 2010. California Fault Lines: Understanding the Causes and Impact of Network Failures. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. ACM, New York, NY, USA, 315–326.

[39] Stefano Vissicchio, Luca Cittadini, and Giuseppe Di Battista. 2015. On iBGP Routing Policies. *IEEE/ACM Trans. Netw.* 23, 1 (Feb. 2015), 227–240. https://doi.org/10.1109/TNET.2013.2296330

[40] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA '16)*. https://doi.org/10.1145/2983990.2984012

[41] Nevin Lianwen Zhang and David Poole. 1996. Exploiting Causal Independence in Bayesian Network Inference. *J. Artif. Int. Res.* 5, 1 (Dec. 1996), 301–328. http://dl.acm.org/citation.cfm?id=1622756.1622765

A APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A.1 Operator Survey

To substantiate the need for a probabilistic network analysis system like NetDice, we conducted an anonymous and voluntary survey amongst network operators in the NANOG mailing list. We received 52 responses over a period of two weeks. 62% (resp. 31%) of participants operate networks connecting more than 10k (resp. 100k) individual users. In the following, we report the three main key findings.

Uncertain events significantly affect forwarding behavior. 56% of operators reported that link and router failures impact forwarding behavior for at least few hours per year, 19% for more than 24h per year. Participants also reported power outages, fiber cuts, and weather events (e.g. floods, storms, tsunamis) to have had noticeable impact in the past. All of which are probabilistic events that can be modeled as failures.

Probabilistic analysis is hard today. While 94% of operators confirmed that they care about probabilistic behaviors, 83% agree or strongly agree that it is currently hard to analyze them. Only 37% use simulators for such analysis and 48% reported not to perform any probabilistic analysis.

The vast majority of the operators would consider using a system such as NetDice. They also note that NetDice's properties (Tab. 1) are practically relevant. 83% would consider using a system that computes the likelihood of a forwarding state under uncertain environments.

A.2 Proofs

PROOF OF LEMMA 5.6. Let $C = E \setminus \mathcal{H}$ be the complement of the set $\mathcal{H}$ returned by Alg. 3. Let NOFAIL be the scenario described by link states L, and FAIL a scenario where additionally, some links in C fail. We prove that the forwarding graph for the considered flow (u, d) is the same in NOFAIL and FAIL.

Stability of pre-processing All routers in $\text{BR}_L \cup \text{RR}_L$ remain connected in FAIL: If there is at least one route reflector in RR_L, Lin. 5 of Alg. 3 makes sure that the routers are connected by hot edges. Otherwise, this is ensured by Lin. 13. Note that there exists a global order on announcements for the pre-processing step TOP3. Hence, even if any border router pruned by TOP3 is disconnected from the routers BR_L in FAIL, the result of TOP3 in Lin. 1 is the same in NOFAIL and FAIL.

Stability of sent announcements Now, we prove that in every round of Alg. 1 (Lin. 2–6), all non-internal routers *send* the same announcements to their peers. The proof works by induction on the rounds.

Base case. In the first round, only the external nodes send announcements. These are independent of link failures and hence identical in NOFAIL and FAIL.

Step case. Consider round i. As an induction hypothesis, assume that in round $i - 1$, all non-internal nodes *sent* the same announcements to their peers in NOFAIL and FAIL. Because non-internal nodes in X remain connected (see above), we can use the induction hypothesis to prove that they *receive* the same announcements from their peers in round i.

Consider any $r \in \text{RR}_L$. For each $b \in \text{BR}_L$, all edges on a shortest path from r to b are in $\mathcal{H}$ (due to Lin. 5). Hence, the IGP cost towards any such b is identical in NOFAIL and FAIL. Because all other attributes involved in the BGP decision process (Tab. 2) do not depend on failures, r selects the same best announcement. Therefore, r sends the same announcements to its peers in round i for NOFAIL and FAIL.

Now, consider any $b \in \text{BR}_L$. We distinguish two cases. *Case (i): b does not send an announcement in round i of NOFAIL.* This can only happen if b receives no announcement at all in NOFAIL (which will also be the case in FAIL), or the best announcement in NOFAIL is received from an internal peer. In the latter case, b will also select an internal announcement in FAIL (due to possibly changed IGP costs, this announcement may however be different from the one selected in NOFAIL). As internal announcements are not re-distributed, b also does not send any announcement in round i of FAIL.

Case (ii): b sends an announcement A in round i of NOFAIL. This can only be the case if A is an external announcement, because internal announcements are not re-distributed. Because of step 5 in Tab. 2, b will for sure also select an external announcement in FAIL. As the preference relation between external announcements according to Tab. 2 does not depend on failures (there is no IGP cost comparison), b will also select and send A in round i of FAIL.

In summary, all non-internal nodes in X send the same announcements to their peers in round i for NOFAIL and FAIL, which concludes the inductive proof.

Stability of selected next hops Next, we show that the selected next hops $\text{NH}_d(v)$ of all decision points $v \in \mathcal{D}$ (see Lin. 6–8 of Alg. 3) are the same in NOFAIL and FAIL. We distinguish two cases.

Case (i): the source u does not select a next hop in NOFAIL. This can only happen if u does not receive any announcements in NOFAIL. Under additional failures, this fact will not change and hence u will also not select a next hop in FAIL. Note that the forwarding graph is empty in this case, therefore there are no other decision points in $\mathcal{D}$.

Case (ii): otherwise. Let $v \in \mathcal{D}$ be an arbitrary decision point, which selects x as next hop in NOFAIL. Further, let A be the selected best announcement at v in NOFAIL in the converged state of Alg. 1 (A has next hop attribute x). In this case, Lin. 10 of Alg. 3 adds the links on a shortest path from v to x to the set $\mathcal{H}$. Hence, the IGP costs between v and x are identical in NOFAIL and FAIL, while the IGP costs towards all other border routers can at most increase due to the failures. Due to our previous argument, v receives the same set of announcements in the converged state of Alg. 3 in NOFAIL and FAIL. Therefore, the BGP decision process (Tab. 2) at v also selects A as the best announcement in FAIL in the converged state, leading to the same next hop x.

Stability of forwarding decisions Next, we prove local stability for all nodes in the forwarding graph. Let y be such a node. We distinguish three cases.

Case (i): y has a configured static route. Lin. 11 of Alg. 3 ensures that y is locally stable under FAIL w.r.t. (u, d).

Case (ii): $y \in \mathcal{D}$. Lin. 10 ensures that the forwarding decision of y for flow (u, d) is the same in NoFail and Fail, because the failure of cold edges cannot change the shortest path towards $\text{NH}_d(y)$.

Case (iii): otherwise. In this case, all of y's parents x in the forwarding graph select the same next hop as y and rely on shortest path routing (otherwise, y would be a decision point by Lin. 7 and Lin. 8). By simple induction, we can recursively apply Lemma 5.4 until we reach a decision point and prove that y is locally stable under Fail w.r.t. (u, d).

Finally, we apply Lemma 5.3 to prove that C is cold. □

Proof of Lemma 7.2. Define $s'' := s[\mathcal{H}^? \leftarrow 1]$ to be the state where all hot edges are up. First, note that the ground states compatible with s' passed to Explore in Lin. 12, together with compat(s''), form a partition of compat(s). Each ground state compatible with s'' or some s' is also compatible with s. Furthermore, sets of compatible ground states for s'' and all instantiations of s' are disjoint because they differ in at least one failed link (Lin. 11) that is locked to be up in Lin. 12 for subsequently considered s'. Last, it is easy to see that each ground state in s is covered by at least one s' or s''.

We prove the lemma using structural induction over the tree of recursive calls to Explore. As induction hypothesis, assume Explore(s') returns $\sum_{L \in \text{compat}(s')} P(\phi \mid L) \cdot P(L)$ in Lin. 12 (if this line is reached).

Note that any $L \in \text{compat}(s'')$ differs from fill(s) at most by failed cold edges. Hence, according to Lemma 7.1, it is

$$P(\phi \mid L = \text{fill}(s)) \;=\; P(\phi \mid L \in \text{compat}(s''))$$

and the value of ρ in line 9 is

$$\hat{\rho} := \textstyle\sum_{L \in \text{compat}(s'')} P(\phi \mid L) \cdot P(L).$$

Using the induction hypothesis and the fact that s' and s'' partition the ground states compatible with s, the returned value in Lin. 12 is

$$\hat{\rho} + \textstyle\sum_{L \in \text{compat}(s) \setminus \text{compat}(s'')} P(\phi \mid L) \cdot P(L),$$

which proves the claim. □