On the Complexity of Network-Wide Configuration Synthesis

Tibor Schneider
ETH Zürich, D-ITET
Zürich, Switzerland
sctibor@ethz.ch

Roland Schmid
ETH Zürich, D-ITET
Zürich, Switzerland
roschmi@ethz.ch

Laurent Vanbever
ETH Zürich, D-ITET
Zürich, Switzerland
lvanbever@ethz.ch

***Abstract*—Configuration Synthesis promises to increase automation in network hardware configuration but is generally assumed to constitute a computationally hard problem. We conduct a formal analysis of the computational complexity of network-wide Configuration Synthesis to establish this claim formally. To that end, we consider Configuration Synthesis as a decision problem, whether or not the selected routing protocol(s) can implement a given set of forwarding properties.**

We find the complexity of Configuration Synthesis heavily depends on the combination of the forwarding properties that need to be implemented in the network, as well as the employed routing protocol(s). Our analysis encompasses different forwarding properties that can be encoded as path constraints, and any combination of distributed destination-based hop-by-hop routing protocols. Many of these combinations yield *NP*-hard Configuration Synthesis problems; in particular, we show that the satisfiability of a set of arbitrary waypoints for any hop-by-hop routing protocol is *NP*-complete. Other combinations, however, show potential for efficient, scalable Configuration Synthesis.

I. Introduction

Configuring IP networks is a complex task nowadays. Starting from high-level requirements, e.g., reachability, isolation, or load-balancing, network operators (often manually) need to come up with low-level configurations specifying the forwarding behavior of possibly hundreds of devices running complex distributed routing protocols. Configuring networks this way is error-prone and often leads to downtimes [1]–[3]. Actually, Alibaba recently revealed that the majority of their network outages resulted from configuration mistakes [4].

Configuration Synthesis aims at preventing such network outages by *automatically* generating correct low-level configurations out of a set of high-level forwarding properties. Using synthesis tools, operators only need to specify *what* should happen to their network traffic, as opposed to *how*. A key challenge of Configuration Synthesis is its scalability. While some systems use domain-specific knowledge to achieve better scalability for specific network topologies, e.g., Propane/AT [5] for data centers, other tools like Zeppelin [6] rely on a best-effort approach without guarantees for the existence or absence of a solution. Current generic synthesis tools, however, cannot support topologies larger than a few tens of routers [7], [8]. Perhaps frustratingly though, it is especially in large networks that operators would benefit the most from synthesis tools.

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

Generally speaking, the complexity of synthesizing correct configurations comes from two ingredients:

(1) the forwarding properties that need to be implemented;
—the space of inputs of the synthesis problem—
(2) the routing protocol(s) we synthesize configurations for.
—the space of outputs of the synthesis problem—

It is a priori not clear how these two dimensions interact. Could it be that some protocols are *easy* to synthesize a configuration for? Is it that some forwarding properties are *hard* to synthesize, independently of the protocols being used? While specific synthesis problems have been shown to be *NP*-hard (e.g. for some protocols [6], [9]), these questions have not been comprehensively answered by the literature thus far.

In this paper, we analyze the computational complexity of network-wide Configuration Synthesis by jointly exploring the space of forwarding properties and routing protocols. We answer to the questions above affirmatively: synthesizing configurations enforcing some forwarding properties is fundamentally hard; while synthesizing configurations for some protocols is easy, in the case of practical forwarding properties.

We explore the space of common routing protocols by considering an abstract *routing algorithm*, that is, any combination of distributed, destination-based hop-by-hop routing protocols. We identify five generic families of routing algorithms ranging from Shortest Path Routing (SPR) protocols to *non-uniform* algorithms, i.e., that allow a separate routing configuration for each destination prefix. While synthesizing bounded integer link weights for SPR (e.g., Open Shortest Path First (OSPF)) is known to be *NP*-hard [9], we show that non-uniform algorithms, such as routing exclusively with the Border Gateway Protocol (BGP), allow for efficient Configuration Synthesis.

Regarding forwarding properties, we identify a set of fundamental path properties ranging from specifying entire paths to specifying waypoint constraints (i.e., mandating that packets should cross a specified device). We show that, perhaps surprisingly, determining the satisfiability of a set of arbitrary waypoint constraints is already *NP*-hard for any hop-by-hop routing algorithm that can be configured to implement any forwarding spanning tree such as OSPF, IS-IS, or BGP (Theorem 3.4). We also observe that while SPR protocols can be configured by inverting the shortest-path computation for a set of fully specified paths, the problem becomes *NP*-hard as soon as some path segments are unspecified.

Specification $\mathcal{S}$ (forwarding properties) / Routing Algorithm $\mathcal{A}$		$\mathcal{S} = \mathcal{P}$ Paths only		$\mathcal{S} = \mathcal{P} \cup V \cup E$ connected Paths and connected waypoints		$\mathcal{S} = \mathcal{P} \cup V \cup E$ Paths and waypoints
Shortest Path Routing, finite bit representation	OSPF or IS-IS	***NP*-Complete**	[9] ⇒	***NP*-Complete**	⇒	***NP*-Complete**
Shortest Path Routing (Section III-A)	—	***P*** $\mathcal{O}((n^2+e)^{2.5})$	[10]	***NP*-Complete**	Th. 3.1 ⇒	***NP*-Complete**
Routing with separate propagation (Section III-B)	BGP + Route Reflection + SPR	***NP*-Hard**	Th. 3.2 ⇒	***NP*-Hard**	⇒	***NP*-Hard**
Hierarchical routing (Section III-C)	Hierarchical BGP + SPR	***NP*-Hard**	Th. 3.3 ⇒	***NP*-Hard**	⇒	***NP*-Hard**
Non-uniform routing (Section III-D)	BGP exclusively	***P*** $\mathcal{O}(d \cdot e)$		⇐ ***P*** $\mathcal{O}(d \cdot e)$	Th. 3.5	***NP*-Hard** Th. 3.4

Fig. 1. The computational complexity of Configuration Synthesis $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ depends on the target routing algorithm $\mathcal{A}$ (rows) and the supported forwarding properties $\mathcal{S}$ (columns, cf. Fig. 2). For problems solvable in polynomial time, its asymptotic complexity is given as a function of the number of routers n, the number of links (edges) e, and the number of destination prefixes d.

In summary, the main contributions of this paper are:

- a taxonomy of the computational complexity (Fig. 1) of network-wide Configuration Synthesis w.r.t. different forwarding properties and routing algorithms;
- a set of proofs showcasing that Configuration Synthesis for common routing algorithms is *NP*-hard;
- the identification of a family of routing algorithms and a subset of forwarding properties that can enable scalable network-wide Configuration Synthesis.

II. Model

We define the problem of network-wide configuration synthesis $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ as a family of algorithmic problems, one for each combination of an abstract routing algorithm $\mathcal{A}$, that is, an abstract representation of the deployed routing protocol(s), and a set of supported forwarding properties $\mathcal{S}$. An instance of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ takes as inputs

- a *network topology* G;
- a set of *routing inputs*, i.e., a set of destination prefixes D, for both internal and externally advertised prefixes, and where each of them is advertised;
- a *network specification* ψ, that is, a set of the supported forwarding properties that need to be satisfied by the converged forwarding state;

and produces as an output a set of parameters for the selected routing algorithm $\mathcal{A}$, for example, link weights or route maps. To analyze the computational complexity of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$, we consider the corresponding decision problem, that is, whether or not the selected routing algorithm $\mathcal{A}$ can implement a given subset of the supported forwarding properties in the network.

To illustrate an instance of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$, consider Fig. 3. On the left, we depict a network topology with two destinations as routing inputs, and the specification that consists of two path properties in green, a set of connected waypoints in blue, and a single arbitrary waypoint in red. This problem has

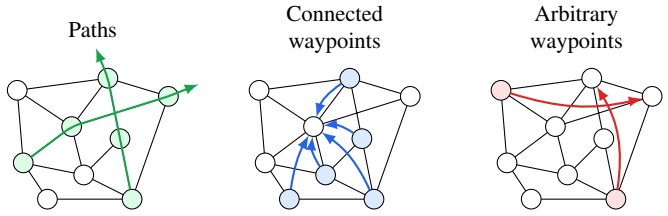

Fig. 2. The forwarding properties $\mathcal{S}$ impact the complexity of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ as they vary in degree of freedom, i.e., number of valid forwarding graphs.

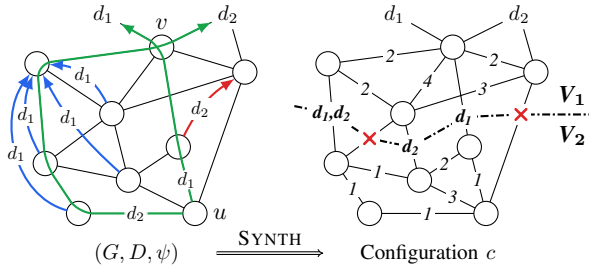

Fig. 3. Configuration Synthesis is the problem of configuring the network to satisfy a specification ψ. The network on the left shows a problem instance of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$ with two destinations $D = \{d_1, d_2\}$. The configuration for OSPF and hierarchical BGP on the right solves this problem using two partitions V_1 and V_2. Edge labels inside partitions are OSPF weights, while those between partitions indicate which destinations are allowed on that edge.

no solution for algorithms such as OSPF as the specification requires two different shortest paths from u to v, one for destination d_1 and one for d_2. Nevertheless, some routing algorithms can solve this problem; Fig. 3 shows such a configuration for hierarchical BGP with OSPF.

In the following, we describe the network model and the two dimensions of $\textsc{Synth}(\mathcal{A}, \mathcal{S})$, namely the different types of network specifications and routing algorithms we consider.

Network Model: We encode the network topology as a directed (multi-)graph $G = (V, E)$, where we denote routers by the set of nodes V and links by directed edges $E \subseteq V \times V$. Each edge $e = (u, v)$ has a source $src(e) = u$ and a destination $dst(e) = v$. A forwarding path $p = \langle e_n, \ldots, e_1 \rangle$ is a sequence of contiguous and loop-free edges with its source $src(p) = src(e_n)$ and destination $dst(p) = dst(e_1)$. We intentionally reverse the path indices to highlight that routing information typically flows in the opposite direction from a forwarding path. We write the concatenation of two paths p and q as $p \circ q$, and denote $p_{k,i} = \langle e_k, e_{k-1} \ldots, e_i \rangle$ to be a sub-path of p. Finally, $\mathcal{P}$ denotes the set of all paths in G.

A. Network Specification (Forwarding Properties)

The network specification can be understood as an invariant on the forwarding paths computed by a distributed routing algorithm, i.e., the converged forwarding state. Hence, operators need to translate their intent on how the network should behave into invariants on that forwarding state—we call these *forwarding properties*. Ideally, operators could use high-level invariants, e.g., requiring all nodes to prefer one neighboring Autonomous System (AS) over another [11], or all traffic from a critical domain to first traverse a firewall. However, the space of such invariants on the forwarding behavior is huge and the support for complex invariants is very limited so far [6], [7].

Instead, we focus our analysis on a set of fundamental forwarding properties that can be expressed as path constraints. These path constraints can then be combined to express the forwarding behavior of an entire network. We can restrict forwarding paths in many ways, ranging from specifying an entire path from a source node s to a destination d (path property) to being agnostic of the path as long as packets eventually reach their destination (reachability) or pass a specified device (waypoint). Since most routing protocols provide reachability by default, synthesizing reachable configurations is often trivial. Hence, we limit our analysis to: path properties that fix the entire forwarding path, and waypoint properties that require a specific node or edge to be traversed by the path.

Formally, we model a specification as a partial function $\psi : (V \times D) \to \mathcal{S}$, where D is the set of destination prefixes. Note that this restricts each node to a single forwarding property per destination. With slight abuse of notation, let

$$\psi(s, d) = \begin{cases} p \in \mathcal{P} & \text{for } \textit{path} \text{ properties,} \\ \omega \in (V \cup E) & \text{for } \textit{waypoint} \text{ properties.} \end{cases}$$

We write $p_{s,d} \models \psi(s, d)$ to indicate that a path $p_{s,d}$ from s to d satisfies the forwarding property $\psi(s, d)$. Due to the constraints of hop-by-hop routing, we assume that any path property $\psi(s, d) = \langle e_n, \ldots, e_1 \rangle$ implies the subsidiary path property $\psi(dst(e_n), d) = \langle e_{n-1}, \ldots, e_1 \rangle$.

Connected Waypoints: In practice, operators rarely need to enforce arbitrarily complex waypoints. Instead, they typically translate higher-level invariants into a set of waypoint properties that relate to each other. For instance, recall the examples from the beginning of this section: modeling AS preferences or enforcing the traversal of a firewall. Both of these invariants can be expressed as a structured set of waypoint constraints by adding the same waypoint to an entire partition of the network. We observe that (1) the shared waypoint is located at the edge of this partition, and that (2) the subgraph induced by the nodes with this waypoint constraint is *connected*. Hence, it is guaranteed that there exists a forwarding tree rooted in ω that satisfies all of these waypoint constraints. More formally:

Definition 2.1: A specification ψ is *connected* if for any waypoint property $\psi(s, d) = \omega$, there exists a path $p \in P$ from s to ω such that, for every node $n \in p$, there exists a property $\psi(n, d) = x$ that implies ω. In other words, any such path from n to d that satisfies $p_{n,d} \models x$ must satisfy $p_{n,d} \models \omega$.

Finally, observe that any hardness result on implementing these basic forwarding properties extends to many more complex network properties. To illustrate this, consider to try finding a configuration that satisfies path preferences $p_a > p_b$ under failure scenarios: A node should prefer the active path p_a as long as it is available, and switch to a backup path p_b only if any link in p_a fails. We claim that synthesizing a configuration that implements these path preferences is *at least as hard* as only synthesizing the path p_a in the first place.

TABLE I
EXAMPLES OF UNIFORM ROUTING ALGORITHMS

Routing Algorithm	Σ	$\oplus$	$\preceq$	ϕ	$\mathcal{C}$
Minimum Hop	$\mathbb{R}^+ \cup \{\infty\}$	$+$	$\leq$	∞	$E \to \{1\}$
OSPF	$\mathbb{R}^+ \cup \{\infty\}$	$+$	$\leq$	∞	$E \to \{1, 2, \ldots, k; \infty\}$
Shortest Path	$\mathbb{R}^+ \cup \{\infty\}$	$+$	$\leq$	∞	$E \to \mathbb{R}^+ \cup \{\infty\}$
Most-Reliable Path	$[0, 1]$	$\cdot$	$\geq$	0	$E \to [0, 1]$

B. Routing Algorithms

We introduce the abstraction of a *routing algorithm* to denote the combination of distributed, destination-based routing protocols that route traffic in a hop-by-hop manner. That is, each node forwards traffic according to its own local decision based solely on the packet's destination. We assume that nodes exchange routing information with their neighbors, and we require that they share at least their preferred route—which is the case for link-state and distance-/path-vector protocols.

We base our model of routing algorithms on Routing Algebra [12], that is, a mathematical tool that generalizes and models the computation of routing protocols. It has proven to be well-suited for protocol and configuration verification, by capturing necessary conditions for optimal routing and eventual consistency [12]–[15]. We transfer this concept to Configuration Synthesis by extending Routing Algebra with a notion of expressivity for an algorithm's configuration.

Formally, we define a *routing algorithm* as a five-tuple

$$\mathcal{A} = (\Sigma, \oplus, \preceq, \phi, \mathcal{C})$$

that contains both a Routing Algebra $(\Sigma, \oplus, \preceq, \phi)$ as well as a description of all possible configurations $\mathcal{C}$. Σ denotes the set

of all weights ordered by the binary relation $\preceq$ and combined with the binary relation $\oplus$. The special symbol $\phi \in \Sigma$ signals that no route towards the destination is known, and is always least-preferred. A configuration $c \in \mathcal{C}$ for a network $G = (V, E)$ with destinations (or prefixes) D is a function

$$c : (E \times D) \to \Sigma$$

that assigns a weight per destination to each edge.

We provide some common examples using this notation in Table I. For instance, we model Shortest Path Routing (SPR) as $\mathcal{A}_{SPR} = (\mathbb{R}^+, +, \leq, \infty, \mathcal{C}_{SPR})$, based on the algebra of positive real numbers. A configuration $c \in \mathcal{C}_{SPR}$ assigns a positive number to each edge. Nodes prefer the path with the smallest *cost*, i.e., the sum of all edge weights along the path, or break ties according to a global edge order.

Path Computation: We model propagated routes as tuples $(d, \sigma, p) \in D \times \Sigma \times \mathcal{P}$, containing a destination d, a cost σ, and a path p. Assuming node v prefers route (d, σ, p) to reach destination d, node v will transform and propagate the route $\big(d,\ c(e,d) \oplus \sigma,\ \langle e \rangle \circ p\big)$ towards u over the edge $e = (u, v)$. In other words, the weight $\sigma \in \Sigma$ announced at $dst(p_{s,d})$ is transformed with the operator $\oplus$ (e.g., added up for SPR) on each edge of the path; i.e., $c(e_n, d) \oplus \ldots \oplus c(e_1, d) \oplus \sigma$ (right-to-left precedence). Further, v will route packets with destination d via the first edge in p. The resulting forwarding graph for a destination d is a directed graph $F_d = (V, E_d)$ with out-degree at most 1, where $E_d \subseteq E$. A forwarding path in F_d from s to d is denoted as $p_{s,d}$ and we write $p_{s,d} \oplus_d \sigma$ to apply the label of each edge along $p_{s,d}$ in propagation order.

Characteristics of Routing Algorithms: Recall that every routing algorithm must have a special symbol ϕ, representing that no route to the destination is known. The following two properties assert that any route will always be preferred over the absence of a route. We assume that both properties hold for all routing algorithms.

Maximality: $a \prec \phi$ for all $a \in \Sigma \setminus \{\phi\}$.
Absorption: $a \oplus \phi = \phi \oplus a = \phi$ for all $a \in \Sigma$.

Other properties that do not hold for all routing algorithms, however, are not as obvious. We call these *characteristics* of routing algorithms. To begin with, we adopt the notions of strict monotonicity and isotonicity from [12]:

Strict Monotonicity: $a \prec b \oplus a$ for all $a, b \in \Sigma \setminus \{\phi\}$.
Strict Isotonicity: $a \prec b \Leftrightarrow c \oplus a \prec c \oplus b\ \forall a, b, c \in \Sigma \setminus \{\phi\}$.

We find that, by themselves, these characteristics are not enough to analyze Configuration Synthesis. For instance, consider Minimum-Hop Routing (MHR) $\mathcal{A}_{MHR}$ that routes traffic according to the path with the least number of hops (modeled as SPR with equal link weights; cf. Table I). Even though both $\mathcal{A}_{MHR}$ and $\mathcal{A}_{SPR}$ satisfy strict monotonicity and isotonicity, $\textsc{Synth}(\mathcal{A}_{MHR}, \mathcal{S})$ degrades to a mere verification (shortest-path computation) of the only possible configuration for MHR, whereas $\textsc{Synth}(\mathcal{A}_{SPR}, \mathcal{S})$ can be solved using Linear Programming (LP).

TABLE II
CLASSIFICATION OF THE MOST COMMON ROUTING PROTOCOL STACKS

Family of Routing Algorithms	Examples
Shortest Path Routing with finite bit representation	OSPF, IS-IS
Shortest Path Routing	SPR, MRPR
Routing with separate propagation	BGP Route Reflection
Hierarchical routing	Hierarchical BGP, BGP Confederations
Non-uniform routing	Exclusively BGP, Static Routes

Consequently, we define the characteristics **C1–C3** to capture the expressivity of a routing algorithm's configuration. Intuitively, a *uniform* routing algorithm routes all traffic identically for all destinations; a *filtering* algorithm allows to configure for each edge whether a node accepts a route on this edge (a: allow), or not (ϕ: deny); and a *linear* routing algorithm is homomorphic to SPR. More formally:

C1 Uniformity: $\mathcal{C}$ requires any configuration $c \in \mathcal{C}$ to have $c(e, d_1) = c(e, d_2)$ for all $e \in E$, $d_1, d_2 \in D$.
C2 Filterability: for some $a \in \Sigma : a \oplus b \prec \phi\ ,\ \forall b \in \Sigma \setminus \{\phi\}$. Further, $\mathcal{C}$ allows both edge weights a and ϕ.
C3 Linearity: $\mathcal{A}$ is strictly monotone, strictly isotone, and there exists a mapping $g : \mathbb{R}_+ \to \Sigma$ such that:
1) $\forall a, b, c \in \mathbb{R}^+ : a = b + c \iff g(a) = g(b) \oplus g(c)$
2) $\forall a, b \in \mathbb{R}^+ : a < b \iff g(a) \prec g(b)$

For instance, consider Most-Reliable Path Routing (MRPR), where routers select the next hop according to the reliability of a path. MRPR can be denoted as $\mathcal{A}_{MRPR} = ([0, 1], \geq, \cdot, 0, \mathcal{C})$. MRPR is strictly monotone and isotone, filtering for any $a \in (0, 1]$ and the function $g(x) = 1/e^x$ is a witness of its linearity.

Finally, we use the introduced notation and terminology to define five generic families of routing algorithms that cover the most common routing protocol stacks (cf. Table II).

Shortest Path Routing with finite bit representation: The most widely used inter-domain routing protocols are SPR algorithms due to their fast convergence upon network failures. They are uniform routing algorithms, because link weights cannot discriminate for different destinations. Furthermore, they are known to be strictly monotone and strictly isotone [12]. Due to practical limitations their link weights are represented with a finite number of bits which makes them non-linear, while a special ∞-label makes them filtering.

Shortest Path Routing: Allowing arbitrary, real-valued link weights makes SPR not only a uniform, but also a linear routing algorithm. This family includes all routing algorithms that are homomorphic to SPR (cf. Lemma 3.1). Note that the family of linear and uniform algorithms can thus only realize forwarding states that satisfy suboptimality (cf. Definition 3.4).

Routing with separate propagation: Internal BGP (iBGP) can be thought of as distributing routing information on an overlay propagation graph. Techniques such as BGP Route Reflection (RR) are motivated by the poor scalability of maintaining iBGP sessions between all pairs of nodes to guarantee that each node eventually receives its preferred route. Nodes perform the actual routing with a different algorithm that is usually linear and uniform, e.g., an SPR algorithm.

Hierarchical routing: Algorithms such as hierarchical BGP partition the network into multiple domains. Each partition routes traffic internally, typically using a linear and uniform routing algorithm but interconnects with neighboring domains using non-uniform routing. This hierarchy allows for forwarding states that do not satisfy suboptimality.

Non-uniform routing: This family encompasses all non-uniform, filtering routing algorithms. They can implement any forwarding spanning tree for each destination prefix independently. Non-uniform routing is equivalent to hierarchical routing if every node constitutes its own partition.

III. Complexity Analysis

Before exploring the complexity of SYNTH($\mathcal{A}, \mathcal{S}$) for the different families of routing algorithms $\mathcal{A}$, we formally define SYNTH($\mathcal{A}, \mathcal{S}$) and discuss how the three kinds of forwarding properties $\mathcal{S}$ relate to each other.

*Definition 3.1 (*SYNTH*($\mathcal{A}, \mathcal{S}$)):* Given a graph $G = (V, E)$, destinations D, and specification $\psi : (V \times D) \to \mathcal{S}$, does there exist a configuration $c \in \mathcal{C}$ for algorithm $\mathcal{A}$ such that c satisfies ψ, i.e., $c \models \psi$?

Observation 3.1: A path property $\psi(s, d) = \langle e_n, \ldots, e_1 \rangle$ can be expressed by a set of connected waypoint properties $\psi(src(e_i), d) = e_i$ for all $1 \leq i \leq n$. Thus, any set of path properties $\psi : (V \times D) \to \mathcal{P}$ can be expressed as a set of connected waypoint properties $\psi : (V \times D) \mapsto V \cup E$. Consequently, any intractability result for SYNTH($\mathcal{A}, \mathcal{S}$) with $\mathcal{S} = \mathcal{P}$ also applies for connected specifications, and a hardness result for a connected specification translates to unrestricted $\mathcal{S} = \mathcal{P} \cup V \cup E$. Likewise, any algorithm solving SYNTH($\mathcal{A}, \mathcal{S}$) for $\mathcal{S} = \mathcal{P} \cup V \cup E$ can also solve the problem if $\mathcal{S}$ is connected, and any algorithm solving SYNTH($\mathcal{A}, \mathcal{S}$) for a connected $\mathcal{S}$ can also solve the problem if $\mathcal{S} = \mathcal{P}$.

A. Shortest Path Routing

This section explores the computational complexity of Shortest Path Routing, as well as its homomorphisms, that is, a routing algorithm $\mathcal{A}$ for which each set of link weights $w \in \mathcal{C}_{SPR}$ for SPR can be transformed to an equivalent configuration $c \in \mathcal{C}$ for $\mathcal{A}$ and vice-versa. We call two configurations equivalent if they will always compute the same forwarding state. We first prove that any linear and uniform routing algorithm $\mathcal{A}$ is a homomorphism of SPR.

Lemma 3.1: Any linear and uniform routing algorithm $\mathcal{A}$ is a homomorphism of SPR.

Proof: Let $\mathcal{A}$ be a linear and uniform routing algorithm. We first show how we transform any configuration $c \in \mathcal{C}$ into equivalent link weights $w \in \mathcal{C}_{SPR}$ for SPR. Since $\mathcal{A}$ is strictly isotone, we can compute the All-Pairs-Shortest-Path (APSP) using the generalized Dijkstra algorithm [12]. While doing so, we construct a system of linear inequalities that we solve using LP to get the equivalent set of link weights w for SPR [16].

Next, we show how to transform any set of link weights for SPR into an equivalent configuration for $\mathcal{A}$. Let $w \in \mathcal{C}_{SPR}$ be a set of link weights for SPR. We construct $c \in \mathcal{C}$ for $\mathcal{A}$ by transforming each weight separately using $g(\cdot)$, i.e., $c(e) = g(w(e))$. Note that we neglect the destination d as both $\mathcal{A}$ and $\mathcal{A}_{SPR}$ are uniform. We will now show that c is equivalent to w by relying on the Linearity of $\mathcal{A}$. Let $p = \langle e_n, \ldots, e_1 \rangle$ be any path in G with $w(p) = w(e_n) + \ldots + w(e_1)$. By repeatedly applying the first rule of **C3**, we get $g(w(p)) = g(w(e_n)) \oplus \ldots \oplus g(w(e_1))$. Finally, let $s, t \in V$, let p^* be the shortest s-t-path, and let $p^- \neq p^*$ be any other s-t-path. $w(p^*) < w(p^-)$ implies that $g(w(p^*)) \prec g(w(p^-))$ by the second rule of g. Hence, any shortest path in G for w is also an optimal path under c as ties are broken identically. This proves Lemma 3.1. ∎

Lemma 3.1 allows us to analyze the entire class of SYNTH($\mathcal{A}, \mathcal{S}$) problems for linear and uniform algorithms $\mathcal{A}$ by simply analyzing SPR. Due to its importance in graph theory, SPR and its inverse problem have been studied extensively in the literature. In the inverse SPR problem, we are given a set of paths $\mathcal{P}$ in a graph G and try to find link weights such that each path $p \in \mathcal{P}$ from $src(p) = s$ to $dst(p) = t$ is the shortest s-t-path in G. Ben-Ameur et al. presented an algorithm to solve the inverse SPR problem in polynomial time using LP [10]. Consequently, SYNTH($\mathcal{A}, \mathcal{S}$) for any linear and uniform $\mathcal{A}$ and $\mathcal{S} = \mathcal{P}$ is solvable in polynomial time using LP in $\mathcal{O}((|V|^2 + |E|)^{2.5})$ time [17].

However, most real-world implementations of SPR like OSPF differ from SPR as they represent link weights using a finite number of bits. It was shown that both finding an exact solution to the discrete inverse SPR problem is *NP*-complete [9], as well as finding a *good* approximation [18].

To the best of our knowledge, the complexity of the inverse SPR problem under waypoint constraints is still an open question. Call et al. have proven the inverse SPR problem *NP*-complete when some edges are required or forbidden on the shortest-path tree towards some destination [19]. Even though this problem relates to SYNTH($\mathcal{A}, \mathcal{S}$) for SPR with waypoints, a key difference is that the problem analyzed by Call et al. cannot require a specific source node to use an edge. Consequently, we analyze the complexity of SYNTH($\mathcal{A}, \mathcal{S}$) for SPR with a connected specification containing waypoints.

Theorem 3.1: Deciding whether there exists a solution to SYNTH($\mathcal{A}, \mathcal{S}$) for any linear and uniform $\mathcal{A}$ and $\mathcal{S} = \mathcal{P} \cup V \cup E$ is *NP*-complete if the specification is connected.

We prove Theorem 3.1 by transforming One-In-Three 3SAT (X3SAT) to SYNTH($\mathcal{A}, \mathcal{S}$), similar to the the proof of Call et al., based on the fact that SPR cannot implement all forwarding graphs. We provide the proof in a technical report[1].

[1]https://doi.org/10.5281/zenodo.6583456

B. Routing with Separate Propagation

We model Internal BGP (iBGP) as an overlay signaling protocol that redistributes external routing information, while an Internal Gateway Protocol (IGP) such as SPR performs the actual routing. In traditional iBGP, all nodes are connected in a full-mesh, leading to performance issues in larger networks [20]. Operators commonly use Route Reflection (RR) to reduce the number of iBGP sessions by selecting a small number of nodes (route reflectors) to redistribute their preferred routes to all other nodes (clients). By reducing the number of such route reflectors, operators can effectively reduce the number of iBGP sessions. Consequently, clients can only select their preferred route if at least one route reflector shares its preference. We consider the configuration of the overlay signaling protocol to be part of the synthesis problem.

We define the Route Reflection problem to identify a set of at most k route reflectors such that each client will receive its required route. This requirement is part of the specification ψ. For instance, path property $\psi(s,d) = p$ requires s to select the route from $dst(p)$ to reach d. Let $\mathcal{R}$ be the set of all routes preferred by at least one node. We can model the RR problem as finding a set of route reflectors $V' \subset V$ with $|V'| \leq k$ such that V' covers all routes in $\mathcal{R}$. Thus, this problem is identical to Minimum Cover (MC), a known *NP*-complete problem [21].

The implementation of BGP RR allows operators to configure an arbitrary overlay signaling graph. This signaling graph is a directed graph $G_{iBGP} = (V, E_{iBGP})$ with route propagation rules as described by Griffin et al. [22]. Operators can reduce the number of iBGP sessions by carefully designing G_{iBGP}. In the following, we thus consider the *iBGP Graph* problem: In order to find a valid propagation graph G_{iBGP} with a minimum number of iBGP sessions (i.e., edges E_{iBGP}), each node must receive its desired route along a valid propagation path in E_{iBGP} where all nodes of that path select the same route.

Definition 3.2 (iBGP Graph): An instance of the iBGP Graph problem is a tuple $I_{iBGP} = (V, D, \mathcal{R}, \tilde{\psi}, k)$, where V is a set of nodes, D is a set of destinations, $\mathcal{R} = (V \times D)$ is a set of routes, $\tilde{\psi} : (V \times D) \mapsto \mathcal{R}$ captures route preferences, and $k \in \mathbb{N}$ is an upper bound on the number of *iBGP* sessions $|E_{iBGP}| \leq k$. A route $(v,d) \in \mathcal{R}$ originates at node v and announces destination d. A node v must select the route $\tilde{\psi}(v,d)$ to reach the destination d. A solution to I_{iBGP} is a directed graph $G_{iBGP} = (V, E_{iBGP})$ with $|E_{iBGP}| \leq k$, such that, for all $v, d \in V \times D$, there exists a path $p \in G_{iBGP}$ from v to the origin of $\tilde{\psi}(v,d)$ along which all nodes prefer $\tilde{\psi}(v,d)$. More formally, we have $\forall u \in p : \tilde{\psi}(u,d) = \tilde{\psi}(v,d)$.

Theorem 3.2: Deciding if there exists a solution to an instance of the iBGP Graph problem is *NP*-complete.

Proof: To begin with, we show that the iBGP Graph problem is in *NP* as a solution to an instance of the iBGP Graph problem can be encoded using $\mathcal{O}(|V|^2)$, i.e., $\mathcal{O}(|E_{iBGP}|)$, bits and verified in polynomial time using a depth-first traversal.

We transform the *NP*-complete problem Vertex Cover (VC) to the iBGP Graph problem. VC is the problem of finding a set of nodes V^* of size $|V^*| \leq k'$ that includes at least one endpoint of each edge in graph $G' = (V', E')$. Let $I_{VC} = (G', k')$ be an instance of VC. Consider Fig. 4 for an example how to construct a basic instance of $I_{iBGP} = (V, D, \mathcal{R}, \tilde{\psi}, k)$ from I_{VC}: Let the nodes be $V = \{x\} \cup \{\hat{e} \mid e \in E'\} \cup \{\hat{v} \mid v \in V'\}$ and let $k = 2|E'| + k'$. Further, we create a destination d_e for each edge $e = (u,v) \in E'$, and let node $\hat{e}$ advertise d_e. We will show how to extend this instance I_{iBGP} such that it is solvable if and only if I_{VC} has a solution.

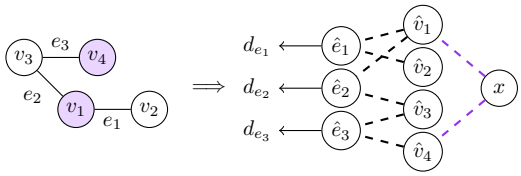

Fig. 4. An instance of the Vertex Cover (VC) problem on the left is transformed to an equivalent iBGP problem on the right. The construction is such that nodes $\hat{v}_1, \hat{v}_2, \ldots$ know routes only to destinations $d_{e_1}, d_{e_2}, \ldots$ that correspond to adjacent edges. Node x must receive all routes from $d_{e_1}, d_{e_2}, \ldots$ by using only k edges to connect x to any node in $\{\hat{v}_1, \hat{v}_2, \ldots\}$ (cf. purple edges). Propagation sessions E_{iBGP} are drawn as dashed lines. A possible solution of VC and of iBGP are drawn in purple.

In the following, consider any edge $e = (u,v) \in E'$. We first extend the construction such that any solution (1) contains the session $(\hat{e}, \hat{u})$ and $(\hat{e}, \hat{v})$, and (2) contains either $(\hat{u}, x)$, $(\hat{v}, x)$, or $(\hat{e}, x)$. Then, we combine those two properties to conclude that I_{VC} is solvable if and only if I_{iBGP} has a solution.

To ensure that any valid solution contains the session $(\hat{e}, \hat{v})$ if $v \in e$, we create a new destination d_v which originates at every node $\hat{u} \neq \hat{v}$. Then, let $\tilde{\psi}(\hat{u}, d_v) = (\hat{u}, d_v)$ and $\tilde{\psi}(\hat{v}, d_v) = (\hat{e}, d_v)$. Session $(\hat{e}, \hat{v})$ has to exist, as $\hat{v}$ must learn its route from $\hat{e}$. This requires any solution of I_{iBGP} to contain at least $2|E'|$ sessions.

Next, we enforce any valid solution to contain either session $(\hat{u}, x)$, $(\hat{v}, x)$, or $(\hat{e}, x)$ for each $e = (u,v) \in E'$. For each node $w \in V \setminus \{\hat{e}, \hat{u}, \hat{v}, x\}$, we let w advertise destination d_e, and select its own route, i.e., $\tilde{\psi}(w, d_e) = (w, d_e)$. Consequently, the propagation path p from d_e towards x can only contain nodes $p \subseteq \{\hat{e}, \hat{u}, \hat{v}, x\}$. As x does not advertise d_e, any solution must contain $(\hat{u}, x)$, $(\hat{v}, x)$, or $(\hat{e}, x)$.

Finally, we show that I_{VC} has a solution if and only if I_{iBGP} is solvable. We first construct a solution V^* to I_{VC} from a solution $G_{iBGP} = (V, E_{iBGP})$ to I_{iBGP}. Due to the first property, E_{iBGP} contains at least $2|E'|$ sessions between any $\hat{e}_x$ and $\hat{v}_y$. Due to the second property, for each edge $e \in E'$, at least one of $\hat{e}$, $\hat{u}$, or $\hat{v}$ are neighbors of x. Now, let V^* be the neighbors of x in G_{iBGP}. If $\hat{e}$ is a neighbor of x for $e = (u,v)$, then simply add u to V^*. Since x must receive $(\hat{e}, d_e)$ for each edge $e \in E'$, V^* forms a vertex cover of G' and $|V^*| \leq k'$. Next, we show how to construct a solution to I_{iBGP} from a solution V^* to I_{VC}. Let E_{iBGP} contain session $(\hat{v}, x)$ for all $v \in V^*$, and session $(\hat{e}, \hat{u})$ for any edge $e \in E'$ with $u \in e$. $G_{iBGP} = (V, E_{iBGP})$ solves I_{iBGP} as $|E_{iBGP}| \leq k$ and V^* is a vertex cover of G', which causes x to receive the route $(\hat{e}, d_e)$ for each edge $e \in E'$. The construction can be done in polynomial time as I_{iBGP} contains $\mathcal{O}(|E'| + |N'|)$ nodes and destinations. This concludes the proof of Theorem 3.2. ■

C. Hierarchical Routing

In hierarchical routing, operators split the network into partitions, each of which behaves as an individual AS. Each partition uses a linear and uniform routing algorithm such as SPR for routing traffic inside the domain, whereas a non-uniform algorithm like BGP exchanges routing information between domains. The introduction of non-uniform link weights increases the number of realizable forwarding states, i.e., the expressivity (cf. Fig. 3).

Formally, a routing algorithm $\mathcal{A}$ is hierarchical if (1) there exists a subset $\Sigma_i \subseteq \Sigma$ for each partition i such that $\Sigma_i, \oplus, \preceq, \phi$ is strictly monotone and isotone, and (2) edges of links within the partition i are uniform and chosen from Σ_i. In other words, for any edge $e \in E$ within partition i,

$$\forall d_1, d_2 \in D: \underbrace{w(e,d_1) \in \Sigma_i}_{\text{monotone and isotone}} \wedge \underbrace{w(e,d_1) = w(e,d_2)}_{\text{uniform}}.$$

Links connecting different partitions may have non-uniform weights chosen from the entire Σ.

If a partitioning of G is given, Configuration Synthesis for $\mathcal{A}$ for this partitioning is *at least as hard as* synthesizing configurations for the IGP used to route traffic within a partition. Otherwise, SYNTH$(\mathcal{A}, \mathcal{S})$ also involves partitioning the network. Notice that assigning every node to its own partition essentially results in a non-uniform routing (cf. Section III-D) as no inter-domain link remains. However, operators usually try to reduce the number of edges between partitions, i.e., reducing the number of cut edges, as these tend to require significantly more maintenance. Thus we require the solution of SYNTH$(\mathcal{A}, \mathcal{S})$ to minimize the number of cut edges.

In the following, we prove that SYNTH$(\mathcal{A}, \mathcal{S})$ for any partitioned $\mathcal{A}$ is intractable for path properties $S = \mathcal{P}$. Since the link weights within a partition are uniform, and strict monotone and isotone, the forwarding paths inside this partition satisfy suboptimality [12]. We thus formulate the Partitioning into Suboptimal Paths (PSP) problem; Given a graph G and a set of paths P, find a partitioning of G such that the sub-paths of P contained within each partition satisfy suboptimality. We prove PSP to be *NP*-complete, thus proving that SYNTH$(\mathcal{A}, \mathcal{S})$ is *NP*-hard for any partitioned $\mathcal{A}$.

Definition 3.3 (PSP): An instance of the PSP problem is $I_{PSP} = (G, P, j)$ with a directed multi-graph $G = (V, E)$ and a set of paths P. The partitioning $V_1, V_2, \ldots$ is a solution to I_{PSP} if and only if it cuts at most j edges, and if the sets of sub-paths P_i of P contained within each partition V_i are suboptimal.

Definition 3.4 (Suboptimality of Paths): A set of paths P satisfies suboptimality if and only if for all pairs of paths $p_1, p_2 \in P$ that have two vertices u, v in common, the two paths share the same sub-path between u and v.

Definition 3.5 (Contained Sub-paths): The set of sub-paths P_i of P contained within V_i is the set of all sub-paths of P containing only edges with both endpoints in V_i.

Theorem 3.3: Deciding if there exists a solution to an arbitrary instance of PSP is *NP*-complete.

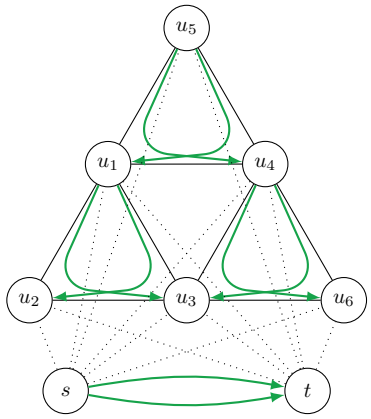

Fig. 5. For constructing an instance of Partitioning into Suboptimal Paths from an instance of Not-All-Equal 3SAT, we create paths for each clause (u_a, u_b, u_c) that violate suboptimality if all nodes are in the same partition. This figure shows an example construction with three clauses (u_1, u_2, u_3), (u_5, u_1, u_4), and (u_4, u_3, u_6).

Proof: To begin with, PSP is in *NP* as a solution to an instance of PSP can be represented using $\mathcal{O}(|V|^2)$ bits and verified in polynomial time by checking the suboptimality of all sub-paths contained within any partition.

We now transform the *NP*-complete problem Not-All-Equal 3SAT (NAE3SAT) to PSP. An instance of NAE3SAT is a tuple $I_{NAE3SAT} = (U, C)$, where U is a set of variables and C is a set of clauses, each containing three variables. A solution is a truth assignment $U_t \subset U$ such that there does not exist a clause for which all variables have an equal assignment [21].

Let $I_{NAE3SAT} = (U, C)$ be an instance of NAE3SAT. We now construct $I_{PSP} = (G, P, j)$. The main idea is to create nodes for each variable and create paths for each clause that satisfy suboptimality only if all three nodes are not part of the same partition (cf. Fig. 5). Let $G = (V, E)$ be a directed multi-graph with the set of nodes $V = U \cup \{s, t\}$ containing all variables and two special nodes s and t. We construct the set of edges E as follows:

- We add two edges $e^1_{s,t}$ and $e^2_{s,t}$ from s to t.
- For each variable $u \in U$, we add edge $e_{s,u}$ from s to u, and $e_{u,t}$ from u to t (dotted lines in Fig. 5).
- For each clause $C_k = (u_a, u_b, u_c) \in C$, we add six edges that form a directed complete graph between u_a, u_b, and u_c called $e^{\square}_{a,b}$, $e^{\square}_{a,c}$, $e^{\square}_{b,a}$, $e^{\square}_{b,c}$, $e^{\square}_{c,a}$, and $e^{\square}_{c,b}$. The placeholder $\square$ in $e^{\square}_{x,y}$ is replaced by the number of already existing edges from u_x to u_y. The first edge from u_x to u_y will be called $e^0_{x,y}$, and the second $e^1_{x,y}$. Hence, edge $e^0_{x,y}$ exists if any clause contains u_x and u_y.

Next, we construct the set of paths P as follows:

- We add two s-t-paths $\langle e^1_{s,t} \rangle$ and $\langle e^2_{s,t} \rangle$ to P.
- For each clause $C_k = (u_a, u_b, u_c) \in C$, we add the two paths $p^1_k = \langle e^0_{a,b}, e^0_{b,c} \rangle$, and $p^2_k = \langle e^0_{a,b}, e^0_{b,c} \rangle$ to P.

Finally, we choose $j = 2 + |U| + 4|C|$. Let P_i be the set of sub-paths of P contained within partition V_i. There are five possibilities for how the partitioning can split the nodes of any clause $C_k = (u_a, u_b, u_c) \in C$ into partitions (cf. Fig. 6):

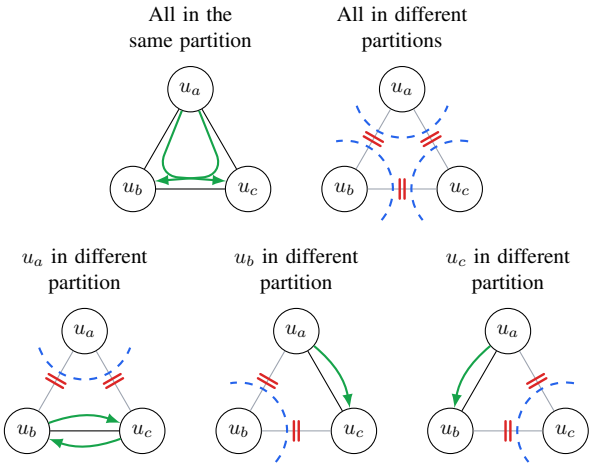

Fig. 6. All five possibilities for partitioning the three nodes of a single clause $C_i = (u_a, u_b, u_c) \in C$ satisfy suboptimality, except if all nodes are in the same partition. This figure shows the resulting paths, as well as the cut edges.

1) All nodes are part of the same partition V_i, violating suboptimality as both paths remain uncut, i.e., $p_k^1, p_k^2 \in P_i$.
2) All three nodes are in different partitions. Hence, the remaining sub-path of p_k^1 and p_k^2 contained within any of the three partitions has zero sizes.
3) Two of the three nodes are in the same partition, resulting in the sub-paths of p_k^1 and p_k^2 having a length of at most 1. Further, for any pair of nodes $u_x, u_y \in V_i$, there can only be one path from u_x to u_y in P_i, which is $\langle e_{x,y}^0 \rangle \in P_i$.

Consequently, any solution to I_{PSP} cannot assign the same partition to all nodes of any clause. Further, if at least one node is part of a different partition for all clauses, then all sub-paths within each partition will satisfy suboptimality.

We will now show that any solution to I_{PSP} partitions the network into exactly two partitions, such that the number of cut edges is less than $j = 2 + |U| + 4|C|$. For this, consider the clause $C_k = (u_a, u_b, u_c) \in C$. Exactly four edges will be cut, unless all three nodes are in different partitions, in which case all six edges are cut (cf. Fig. 6). Further, we know that s and t are part of two different partitions, as $\langle e_{s,t}^1 \rangle$ and $\langle e_{s,t}^2 \rangle$ are not suboptimal. Consequently, the optimal partitioning that solves I_{PSP} will use two partitions and will cut precisely $j = 2 + |U| + 4|C|$ edges; two edges connecting s and t, i.e., $e_{s,t}^1$ and $e_{s,t}^2$, then either edge $e_{s,u}$ or $e_{t,u}$ for each node $u \in U$, and finally, four edges for each clause.

We now show that $I_{NAE3SAT}$ has a solution if and only if I_{PSP} is solvable. Let $V_1, V_2, \ldots$ be a solution to I_{PSP}. The truth assignment $U_t = U \cap V_1$ is also a solution to $I_{NAE3SAT}$ because (1) $V_2 = V \setminus V_1$ since any solution to I_{PSP} has precisely two partitions, and (2) for each clause, the three nodes cannot all be in the same partition. Similarly, let the truth assignment U_t be a solution to $I_{NAE3SAT}$. Then, the partitioning $V_1 = U_t \cup \{s\}$ and $V_2 = V \setminus V_1$ is a solution to I_{PSP}, as the nodes of any clause must not have equal value, i.e., are not all in the same partition. The construction is polynomial in time, as there are $\mathcal{O}(|U| + |C|)$ edges in G. This proves Theorem 3.3. ■

D. Non-Uniform Routing

Some routing protocols, most notably BGP, can be configured without limitations. Attributes of advertisements, i.e., its cost, can be changed arbitrarily for each destination prefix, or the route can even be blocked entirely. A practical synthesis tool already exist [5]. However, the computational complexity of configuring a network only with BGP is yet to be analyzed. We model such protocols as filtering and non-uniform algorithms. The filtering property allows nodes to prefer specific edges over others, while the non-uniformity enables the network to apply this preference separately per destination. Consequently, we can design any forwarding forest F_d for each destination $d \in D$ separately by choosing proper link weights:

Lemma 3.2: Let $G = (V, E)$ be a graph with destinations D and let $\mathcal{A}$ be a non-uniform and filtering routing algorithm. There exists a configuration $c \in \mathcal{C}$ for $\mathcal{A}$ that results in any spanning forest $F_d = (V, E_d)$ for destination $d \in D$.

Proof: We construct $c \in \mathcal{C}$ which computes the spanning forest $F_d = (V, E_d)$. Let a be a weight such that $a \oplus b \prec \phi$ for all $b \in \Sigma \setminus \{\phi\}$ as required by **C2**. We construct c as follows:

$$c(e, d) = \begin{cases} a & \text{if } e \in E_d \\ \phi & \text{if } e \notin E_d \end{cases}$$

We now show that c computes F_d. Let $p = \langle e_n, \ldots, e_1 \rangle$ be a path, and let $e_i \notin E_d$ be any edge in p which is not in F_d. Further, let p^* be the path on F_d from $s = src(p)$ to its root. We will now show that node s will prefer $p^* \oplus_d \sigma_1$ over $p \oplus_d \sigma_2$ for any $\sigma_1, \sigma_2 \in \Sigma \setminus \{\phi\}$. $p^* \oplus_d \sigma_1 = a \oplus \cdots \oplus a \oplus \sigma_1 \prec \phi$, as any edge in p^* is on T_d. However, by absorption, $p \oplus_d \sigma_b = \cdots \oplus \phi \oplus \cdots \oplus \sigma_b = \phi$. Consequently, node s will always prefer $p^* \oplus_d \sigma_1$ over $p \oplus_d \sigma_2$. This proves Lemma 3.2. ■

Notice that most non-uniform routing algorithms $\mathcal{A}$ can implement any spanning forest for each destination without filtering routes. For example, BGP can implement any spanning forest for destination d by assigning routes learned over edge $e \in E_d$ a high local preference.

We now consider specifications containing arbitrary (non-connected) waypoints. We first focus on the Spanning Tree with Waypoints (STW) problem: find a forwarding spanning tree F_d for a single destination d that satisfies a set of waypoint properties. By proving STW to be *NP*-complete, we also show that SYNTH$(\mathcal{A}, \mathcal{S})$ is *NP*-hard as we can transform STW to SYNTH$(\mathcal{A}, \mathcal{S})$ by only considering a single destination d. Perhaps more surprisingly, Theorem 3.4 proves SYNTH$(\mathcal{A}, \mathcal{S})$ to be *NP*-hard not only for non-uniform and filtering algorithms but also for other algorithms that can implement any spanning tree for a single destination, i.e., any filtering algorithm.

Definition 3.6 (Spanning Tree with Waypoints): An instance of the STW problem is a tuple $I_{STW} = (G, r, \tilde{\psi})$, where $G = (V, E)$ is a graph, $r \in V$ is a root node, and $\tilde{\psi} \in (V \times V)$ is a set of pairs of nodes. A solution to the STW problem is a spanning tree $T = (V, E_T)$ rooted at r such that for all pairs $(s, \omega) \in \tilde{\psi}$, the path from s to r traverses node ω.

Theorem 3.4: Deciding whether an arbitrary instance of STW is solvable is *NP*-complete.

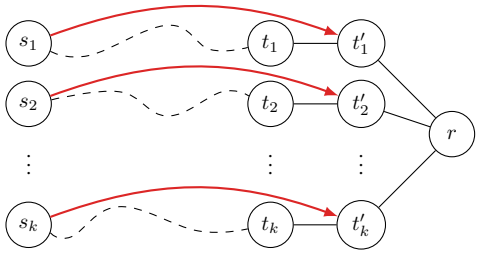

Fig. 7. For constructing an instance of STW from DCP, we add the root node r, as well as nodes t'_i, such that a spanning tree from a solution to STW can be formed by connecting each t_i to t'_i, and t'_i to r. The arbitrary waypoint properties are shown as red arrows.

Proof: To begin with, we show that STW is in *NP*. This is because a solution can be represented with $\mathcal{O}(|E|)$ bits and be verified in polynomial time by simply constructing all paths and checking the waypoint properties $\tilde{\psi}$.

We now transform Disjoint Connecting Paths (DCP) to STW. DCP is the *NP*-complete problem of finding a set of k vertex disjoint paths in a graph $G = (V, E)$ that connect each of the k pairs in $\tilde{\psi} \in V \times V$ [21].

Let $I_{DCP} = (G, \tilde{\psi})$ be an instance of the DCP problem. We construct a problem instance $I_{STW} = (G', r, \tilde{\psi}')$ which is solvable if and only if I_{DCP} has a solution. Let $G' = (V', E')$ be a graph constructed from G, adding the root node r and, for each pair $(s_i, t_i) \in \tilde{\psi}$, the node t'_i that is only connected to t_i and r. Finally, we use the waypoint properties that s_i must reach r via t'_i, i.e., $\tilde{\psi}' = \{(s_i, t'_i) \mid (s_i, t_i) \in \tilde{\psi}\}$. We provide an example of such a graph in Fig. 7.

We now show how to construct a solution to I_{DCP} from a solution to I_{STW} in polynomial time. Let $T = (V, E_T)$ be a spanning tree that solves I_{STW}, which means that all waypoint properties are satisfied. Since T is a tree, there can only be one path from s_i to r, which we call p'_i. Each path p'_i must end in $\langle (t_i, t'_i), (t'_i, r) \rangle$, because $t'_i \in p'_i$ and t'_i has precisely two neighbors, t_i and r. r is the root node and can therefore not be a child of t_i. Further, any two paths in a tree from two nodes u and v to the root can merge but not separate since any node has at most one parent. Hence, all paths from s_i to r are vertex disjoint. As a consequence, the set of sub-paths from p'_i up to node t_i form a solution to I_{DCP}.

Finally, we use a solution to I_{DCP} to construct a spanning tree $T = (V, E_T)$ that solves I_{STW} in polynomial time. Let $p_1, p_2, \ldots$ be a set of vertex-disjoint paths, where $\forall (s_i, t_i) \in \tilde{\psi}$, p_i is a path from s_i to t_i. We first add all edges in all paths to E_T, i.e., $\forall i : p_i \subseteq E_T$. We then set the parent of each t_i to t'_i, and the parent of t'_i to r, i.e., $(t_i, t'_i) \in E_T$ and $(t'_i, r) \in E_T$. Finally, we connect all remaining nodes arbitrarily to r using a breath-first traversal that preserves the tree. T is a spanning tree because (1) each node is connected to the root node, and (2) every node except r will have precisely one parent as every node is traversed at most once by all paths p_i. The spanning tree satisfies all waypoints in $\tilde{\psi}'$ because each t_i has t'_i as its parent. This concludes the proof of Theorem 3.4. ■

Require: $G = (V, E), d, \rho, \psi$
Ensure: $F_d \models \psi$

```
1:  F_d ← ∅
2:  V_0 ← {v ∈ ρ(d) | ⟨⟩ ⊨ ψ(v, d)}
3:  q ← {(dst(e), ⟨e⟩) | ∃v ∈ V_0 : src(e) = v}
4:  while q ≠ ∅ do
5:      (v, p) ← POP(q)
6:      if v ∉ F_d ∧ p ⊨ ψ(v, d) then
7:          F_d(v) ← e
8:          PUSH(q, {(dst(e), p ∘ ⟨e⟩) | e ∈ E : src(e) = v})
9:      end if
10: end while
11: if ∃v ∉ V_0 : F_d(v) = ∅ then
12:     raise UNSAT
13: end if
```

Fig. 8. Algorithm for generating a spanning forest F_d for destination d that satisfies the specification ψ. The algorithm is based on a breath-first traversal. In Line 3 we prepare the queue to contain all outgoing edges of nodes that advertise a route towards d and are allowed by ψ to select its own route.

Connected Waypoints: In the following, we present an algorithm to synthesize a configuration for any filtering and non-uniform routing algorithm $\mathcal{A}$ in polynomial time, satisfying a connected specification. The algorithm performs a breath-first traversal for each destination $d \in D$ to build the spanning forest F_d. This traversal starts at the set of nodes $\rho(d) \subseteq V$ that originate destination d. The algorithm explores node v only if the current path p satisfies the property $p \models \psi(v, d)$. We generate a configuration c for destination d using Lemma 3.2 and the constructed spanning forest F_d. The algorithm shown in Fig. 8 has running time $\mathcal{O}(|D| \cdot |E|)$, as it performs one breath-first traversal (with complexity $\mathcal{O}(|E|)$) for each $d \in D$.

Theorem 3.5: For any graph $G = (V, E)$, connected specification ψ, and destination $d \in D$ announced at $\rho(d) \subseteq V$, the algorithm in Fig. 8 yields a valid spanning forest for d that satisfies ψ if and only if such a spanning forest exists.

Proof: We first show that any spanning forest $F_d \models \psi$ constructed by Fig. 8 satisfies ψ. Since a parent is assigned to any node only if the resulting path satisfies its property (see Line 6) and if it has yet no parents (see Line 6), the resulting spanning forest $F_d \models \psi$. This proves that if no valid spanning forest exists, then the algorithm will not terminate properly.

What is left to show is that the algorithm in Fig. 8 will find a valid solution if there exists a spanning forest $F^*_d \models \psi$. Let F^*_d be such a spanning forest. We now need to show that the algorithm cannot raise `Unsat`, as it would otherwise return a spanning forest $F_d \models \psi$. Thus, at Line 11, there exists a node $v \notin V_0$ who has no parents in F_d. Let V_x be the set of such nodes, and pick any $v \in V_x$. Due to Definition 2.1, there exists a path p starting at v towards a root in F_d along with every node has a property that complies with $\psi(v, d)$. Let p be such a path. Let e be the last edge on p for which $src(e) \in V_x$, but $dst(e) \notin V_x$. Due to Definition 2.1, the sub-path of p starting from $dst(e)$ must also comply with the property $\psi(src(e), d)$. This leads to the contradiction, as $src(e) \in V_x$ cannot hold when $dst(e) \notin V_x$ because the edge e is pushed onto the queue in Line 8. This concludes the proof of Theorem 3.5. ■

IV. Related Work

Before concluding this paper, we summarize the literature related to formal methods applied to network management.

Network Configuration Synthesis: In recent years, the literature has proposed several Configuration Synthesis systems. Many aim to be as general as possible, allowing operators to deploy them in a wide range of networks [7], [8], [23]. Unfortunately, these systems do not scale to large networks. For instance, the state-of-the-art SMT-based system NetComplete takes around 6 hours to synthesize configurations for 64 nodes [7]. Other synthesizers specifically target data centers, allowing these systems to utilize domain-specific knowledge to improve their scalability with a modular synthesis approach [5], [24]. While being very effective in synthesizing configurations for data-center networks, their approach has yet to be generalized for other Internet networks, e.g., transit networks. Finally, some systems follow a best-effort approach while introducing static routes for fixing issues [6], which enables much better scaling than for SMT-based approaches; however, without any guarantees to find a solution and at the cost of network reliability, e.g., upon failures. In contrast, this paper does not propose a system to facilitate configuration synthesis in a specific scenario, but explains the limitations of previous work by studying the computational complexity of Configuration Synthesis.

Network Configuration Verification: Configuration Verification is the task of verifying whether a network configuration correctly implements a set of network properties. Some verification systems use SMT-based solvers to check both forwarding and control-plane properties under different failure scenarios [25]–[28]. Modern approaches show great scalability by relying on a modular and concurrent design [29], [30]—this promising insight may also apply to Configuration Synthesis. However, even though verification and synthesis seem very similar, there are critical differences between them. For example, in verification, one goal is to analyze unstructured, human-written, and complex configurations that grew over the years, whilst synthesis can limit itself to a well-defined structure in the generated configurations. Thus, tractability results presented in this paper do not immediately apply to Configuration Verification.

Complexity of Configuration Synthesis: The computational complexity of individual aspects of Configuration Synthesis has already been analyzed in the literature. Several studies have focused on inverting Shortest Path Routing [9], [10], [16], [18]. In this paper, we analyze the problem of computing link weights for SPR to realize a set of waypoint properties. Other aspects, like adding or removing individual iBGP sessions [31], or partitioning the network to comply with BGP propagation rules [6], were already shown to be intractable. In contrast to previous work, we analyze both dimensions of Configuration Synthesis that influence its complexity—the routing algorithm and the forwarding properties.

V. Conclusion

This paper has analyzed the computational complexity of Configuration Synthesis by exploring the spaces of both routing protocols and forwarding properties. We identify a set of fundamental path-based forwarding properties that (1) capture Configuration Synthesis to transition from being computationally tractable to *NP*-hard and (2) allow our results to generalize to many higher-level requirements. Further, we find that synthesizing configurations satisfying arbitrary waypoint constraints is *NP*-hard for any hop-by-hop routing protocol.

We model the routing protocols and their combinations as abstract *routing algorithms*, and characterize these routing algorithms according to properties of their routing computation. We find the complexity of Configuration Synthesis to depend on the expressivity of the routing algorithm, i.e., how freely its parameters can be tuned. For instance, we can synthesize link weights for SPR that satisfy path properties in polynomial time only if these weights can take any positive real number. Similarly, synthesizing configurations for uniform routing protocols (i.e., protocols using a shared routing configuration for all destination prefixes) is generally *harder* than for non-uniform protocols, as the latter can realize forwarding graphs for each destination independently. However, increasing the expressivity of an algorithm's configuration does not necessarily imply that its synthesis becomes *easier*. To see that, consider SPR and hierarchical BGP. Hierarchical BGP strictly exceeds the expressivity of SPR as SPR can only realize a forwarding state if it satisfies the suboptimality principle. However, while deciding whether there exists a configuration for SPR to implement a set of forwarding properties is computationally tractable, the same problem for hierarchical BGP is *NP*-hard.

While we analyzed the complexity of Configuration Synthesis for the most common routing algorithms, we could not analyze *all* of them. For example, the complexity of synthesizing configurations for Widest-Path Routing (e.g., EIGRP), as well as other algorithms that lack strict monotonicity and strict isotonicity, is still an open question.

We show that Configuration Synthesis for traditional routing protocol stacks, such as OSPF and hierarchical BGP, is *NP*-hard and thus impractical for large networks. Our results suggest non-uniform routing algorithms to be a promising alternative for designing new synthesis tools. Non-uniform algorithms have seen widespread adoption in data-center networks already, for which scalable synthesis tools exist; however, they see little adoption in other Internet networks thus far as there are many challenges left to enable complete network automation in general networks.

Acknowledgements

We thank João Sobrinho for his helpful feedback, as well as our shepherd Ryan Beckett and the anonymous reviewers for their insightful comments. The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809.

REFERENCES

[1] R. King, "Here's Why Amazon's Cloud Suffered a Meltdown This Week," Mar. 2017, accessed: 2021-03-09. [Online]. Available: fortune.com/2017/03/02/amazon-cloud-outage/

[2] M. Locklear, "Google accidentally broke the internet throughout Japan," Engadget, Aug. 2017, accessed: 2021-03-09. [Online]. Available: engadget.com/2017-08-28-google-accidentally-broke-internet-japan.html

[3] D. Madory, "Facebook's historic outage, explained," Oct. 2021, accessed: 2021-03-09. [Online]. Available: kentik.com/blog/facebooks-historic-outage-explained/

[4] H. H. Liu, X. Wu, W. Zhou, W. Chen, T. Wang, H. Xu, L. Zhou, Q. Ma, and M. Zhang, "Automatic Life Cycle Management of Network Configurations," in *ACM SelfDN*, Budapest, Hungary, 2018.

[5] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Network configuration synthesis with abstract topologies," *SIGPLAN Not.*, vol. 52, no. 6, p. 437–451, Jun. 2017.

[6] K. Subramanian, L. D'Antoni, and A. Akella, "Synthesis of fault-tolerant distributed router configurations," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 1, Apr. 2018.

[7] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "NetComplete: Practical network-wide configuration synthesis with autocompletion," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 579–594.

[8] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Aed: Incrementally synthesizing policy-compliant and manageable configurations," in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 482–495.

[9] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing ospf weights," in *Proceedings IEEE INFOCOM 2000. conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064)*, vol. 2. IEEE, 2000, pp. 519–528.

[10] W. Ben-Ameur and E. Gourdin, "Internet routing and related topology issues," *SIAM Journal on Discrete Mathematics*, vol. 17, no. 1, pp. 18–49, 2003.

[11] L. Gao and J. Rexford, "Stable internet routing without global coordination," *IEEE/ACM Trans. Netw.*, vol. 9, no. 6, p. 681–692, Dec. 2001.

[12] J. L. Sobrinho, "Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*, vol. 2. IEEE, 2001, pp. 727–735.

[13] ——, "Network routing with path vector protocols: Theory and applications," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03, New York, NY, USA, 2003, p. 49–60.

[14] ——, "Correctness of routing vector protocols as a property of network cycles," *IEEE/ACM Trans. Netw.*, vol. 25, no. 1, p. 150–163, Feb. 2017.

[15] J. L. Sobrinho and M. A. Ferreira, "Routing on multiple optimality criteria," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 211–225.

[16] S. Das, O. Egecioglu, and A. El Abbadi, "Anónimos: An lp-based approach for anonymizing weighted social network graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 4, pp. 590–604, 2012.

[17] P. M. Vaidya, "Speeding-up linear programming using fast matrix multiplication," in *30th annual symposium on foundations of computer science*. IEEE Computer Society, 1989, pp. 332–337.

[18] A. Bley, "Inapproximability results for the inverse shortest paths problem with integer lengths and unique shortest paths," *Networks: An International Journal*, vol. 50, no. 1, pp. 29–36, 2007.

[19] M. Call and K. Holmberg, "Complexity of inverse shortest path routing," in *International Conference on Network Optimization*. Springer, 2011, pp. 339–353.

[20] E. Chen, T. J. Bates, and R. Chandra, "BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP)," RFC 4456, Apr. 2006.

[21] M. R. Garey and D. S. Johnson, *Computers and intractability: a Guide to the Theory of NP-Completeness*. freeman San Francisco, 1979, vol. 174.

[22] T. G. Griffin and G. Wilfong, "On the correctness of ibgp configuration," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 17–29.

[23] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 261–281.

[24] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 328–341.

[25] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Scalable verification of border gateway protocol configurations with an smt solver," *SIGPLAN Not.*, vol. 51, no. 10, p. 765–780, Oct. 2016.

[26] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 155–168.

[27] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker, "Nv: An intermediate language for verification of network control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 958–973.

[28] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 750–764.

[29] A. Tang, R. Beckett, K. Jayaraman, T. Millstein, and G. Varghese, "Lightyear: Using modularity to scale bgp control plane verification," 2022.

[30] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker, "Modular control plane verification via temporal invariants," 2022.

[31] S. Vissicchio, L. Cittadini, L. Vanbever, and O. Bonaventure, "ibgp deceptions: More sessions, fewer routes," in *2012 Proceedings IEEE INFOCOM*. IEEE, 2012, pp. 2122–2130.