Yale University
Department of Computer Science

Design Principles of Policy Languages for Path-Vector Protocols

Timothy G. Griffin Aaron D. Jaggard
Vijay Ramachandran

YALEU/DCS/TR-1250
April 2004

This work was partially supported by the U.S. Department of Defense (DoD) University Research Initiative (URI) program administered by the Office of Naval Research (ONR). A shortened form of this work has been published as a conference paper [8].

Design Principles of Policy Languages for Path-Vector Protocols

Timothy G. Griffin* Aaron D. Jaggard† Vijay Ramachandran‡

Abstract

BGP is unique among IP-routing protocols in that routing is determined using semantically rich routing policies. However, this expressiveness has come with hidden risks. The interaction of locally defined routing policies can lead to unexpected global routing anomalies, which can be very difficult to identify and correct in the decentralized and competitive Internet environment. These risks increase as the complexity of local policies increase, which is precisely the current trend. BGP policy languages have evolved in a rather organic fashion with little effort to avoid policy-interaction problems. We believe that researchers should start to consider how to *design* policy languages for path-vector protocols in order to avoid routing anomalies while obtaining desirable protocol properties. We take a few steps in this direction by identifying the important dimensions of this design space and characterizing some of the inherent design trade-offs. We do this in a general way that is not constrained by the details of BGP.

This work was partially supported by the U.S. Department of Defense (DoD) University Research Initiative (URI) program administered by the Office of Naval Research (ONR). A shortened form of this work has been published as a conference paper [8].

*Intel Research Laboratory at Cambridge, Cambridge, UK. E-mail: tim.griffin@intel.com. This work was done while at AT&T Labs – Research, Florham Park, NJ, USA.

†Dept. of Mathematics, Tulane University, New Orleans, LA, USA. E-mail: adj@math.tulane.edu. Partially supported by ONR Grant N00014-01-1-0795 and by ONR Grant N00014-01-1-0431. This work was done while at the Dept. of Mathematics, University of Pennsylvania, Philadelphia, PA, USA.

‡Dept. of Computer Science, Yale University, New Haven, CT, USA. E-mail: vijayr@cs.yale.edu. Partially supported by a 2001–2004 U.S. DoD National Defense Science and Engineering Graduate (NDSEG) Fellowship and by ONR Grant N00014-01-1-0795.

1 Introduction

The Border Gateway Protocol (BGP) is the dynamic routing protocol used to connect autonomously administered networks on the Internet [12, 21, 25]. BGP's main task is to establish and maintain best-effort connectivity, even in the face of large-scale network outages. This contrasts with other, more familiar IP-routing protocols such as OSPF and IS-IS, whose main task is to establish and maintain connectivity *within* a single administrative domain [14].

BGP is unique among IP-routing protocols in that routing is determined using semantically rich routing policies. It is important to note that the languages and techniques for specifying BGP routing policies are not actually a part of the protocol. The BGP specification (RFC 1771 [21]) merely describes the low-level binary formats of BGP update messages, the intended meaning of the fields included in update messages, and the correct operation of a BGP-speaking router. On the other hand, routing-policy languages have been developed by router vendors and have evolved through interactions with network engineers in an environment lacking vendor-independent standards. Vendors typically provide hundreds of special commands for use in the configuration of BGP policies. In addition, BGP communities (RFC 1997 [3]) allow policy writers to selectively attach tags to routes and use these to signal policy information to other BGP-speaking routers. Routing policies can then condition their behavior on the presence or absence of specific community values. These developments have more and more given the task of writing BGP configurations aspects associated with open-ended programming. This allows network operators to encode complex policies in order to address unforeseen situations and has opened the door for a great deal of creativity and experimentation in routing policies.

However, this rich expressiveness has come with hidden risks. The interaction between locally defined routing policies can lead to unexpected global routing anomalies such as nondeterministic routing and protocol divergence [9, 26]. If the interacting policies causing such anomalies are defined in separate, autonomously administered networks, then these problems can be very difficult to debug and correct. For example, the setting of an attribute in one autonomous system to implement "cold-potato routing" can cause protocol divergence in a neighboring autonomous system [4, 18]. We suspect that such problems will only become more common as BGP continues to evolve with richer policy expressiveness. For example, extended communities [20] provide an even more flexible means of signaling information within and between autonomous systems than the original definition [3] did. At the same time, applications of communities by network operators are evolving to address complex issues of interdomain traffic engineering [2].

We believe that the root cause of "BGP-configuration problems" is a lack of design for the policy languages that are used to configure this protocol. BGP policy languages have evolved in a rather organic fashion with little or no effort made to avoid policy-interaction problems. We believe that researchers should start to consider how to *design* policy languages and path-vector protocols that together avoid such risks and yet retain other desirable features. We take a few steps in this direction by identifying the important dimensions of this design space and characterizing some of the inherent design trade-offs. We do this in a general way that is not constrained by the details of BGP. As a result, our framework may offer guidance not only in the analysis of proposals to correct or extend BGP but also in the analysis of other BGP-like protocols such as a version of BGP supporting Virtual Private Networks [22], Telephony Routing over IP (TRIP) [23], and of various proposals for interdomain routing of optical paths [19, 27].

1.1 Overview of the Design Space

We feel that our main contribution is in the identification of the *design goals* of policy languages and path-vector protocols. In addition, we formalize these goals and path-vector implementations in a way that allows inherent trade-offs to be rigorously characterized.

We identify six important design goals for any path-vector protocol and policy language:

Expressiveness. From the perspective of a network operator, we desire policy languages that are as *expressive* as possible. For example, shortest-path routing is not expressive enough for the requirements of current interdomain routing because it is unable to capture the "natural" routing conditions arising from the pervasive economic roles of customer, provider, and peer [15, 16]. The challenge then is to design policy languages that are as expressive as possible, and yet not so expressive that other design goals are sacrificed.

Robustness. We require predictability, *i.e.*, that any nondeterminism in routing policies is not the result of unwanted policy interactions, and the existence of a routing solution which is always found by the protocol (this prevents protocol divergence). Furthermore, we insist that the same is true of any configuration that results from any combination of link and node failures in the network. The goal of robustness is the primary constraint on the expressive power of a policy language; we are generally uninterested in non-robust policies.

Autonomy. Network operators often require a high degree of *autonomy* when defining routing policies. We may have a good intuition about what this means—that policy writers are given wide latitude in defining policies that reflect their own interests and not the interests of their neighbors. Here, generalized autonomy will mean the ability to define a partition on routes and then rank the partition classes arbitrarily. Operationally, autonomy is important because it isolates an autonomous system from policy changes occurring in other (neighboring or distant) autonomous systems. Without a high degree of autonomy, network operators would have to continually "tweak" their policies to compensate for unseen changes made to policies elsewhere.

In addition to a generalized definition, we present one notion of autonomy important for BGP—*autonomy of neighbor ranking*—that allows policy writers to classify neighbors and set route preferences in accordance with this classification. This type of autonomy is required for a BGP policy language to support policies compatible with present-day commercial realities of the Internet.

Protocol Transparency. Many "obvious" approaches to achieving very expressive and robust systems involve a high cost; they add machinery that is invisible to policy writers to the underlying path-vector system. What is lost is *protocol transparency*—the ability of network operators to understand the semantics of policies they write. If the protocol itself is allowed to dynamically modify the input policies (in order to ensure robustness, for example), then it may become very difficult, if not impossible, to maintain and debug routing policies.

Global Consistency. One way to achieve robustness is to implement a mechanism enforcing a global-consistency constraint that guarantees robustness. This constraint could be enforced in any number of ways, including an additional protocol or set of protocols, by convention, by regulation, by economic incentives, or by some combination of methods. Of course, the easier such a constraint is to check, the better. We note that in the current Internet, there is no global-consistency checking of BGP policies.

Policy Opaqueness. This design goal measures the degree to which details of routing policies are to be kept private or hidden from those outside of a routing domain (the term is from Geoff Huston [17]). Full policy opaqueness is, of course, in direct conflict with any sort of global-consistency enforcement. Therefore, the design challenge is to find a happy medium that allows for the exposure of just enough information to ensure robustness while at the same time allowing for a sufficient amount of information hiding to satisfy policy writers.

Our formalization starts with defining three distinct components of any path-vector protocol: the underlying path-vector system, the policy language, and any global consistency assumptions about the network. The path-vector system should be thought of as the low-level means of carrying messages between systems, much like RFC 1771. Section 2 presents a definition for path-vector systems that formalizes the information that nodes exchange, various restrictions on nodes' behavior, and the way that protocols mediate interactions between nodes. As we define various components, we illustrate them with a running example that models BGP. Additional examples are given in Section 3.

We separate the definition of a path-vector system from the definition of a policy language: a policy language is a high-level declaration of how the attributes describing a route change when the route is exchanged between neighbors. Section 2.3 defines the intended role of policy languages in path-vector-system configuration.

The notions of expressiveness and robustness are formalized in Sections 4 and 5. For both we employ the Stable Paths Problem (SPP) [9] as a semantic model of path-vector systems. We identify one class of robust systems as our target for expressiveness (Definition 5.4 and Theorem 5.10). Autonomy and transparency are formalized in Sections 6.1 and 6.2. Policy opaqueness is briefly discussed in Section 6.4, while global constraints are considered in Section 7.

Besides the more obvious trade-offs already mentioned, we identify several more subtle ones:

1. Any system with a policy language that is maximally expressive but has no global constraint must give up either autonomy of neighbor ranking or transparency (or both) (Theorem 6.9).

2. Any autonomous, transparent, and robust system with a policy language at least as expressive as shortest-path routing must have a non-trivial global constraint (Theorem 7.4).

These results tell us that, if we seek to design expressive policy languages that are transparent, autonomous, and robust, then we must consider the global constraint as an integral part of the design. Indeed, current path-vector protocols may succeed in part because of assumptions about the global network; our framework highlights the importance of this component of design.

Figure 1 illustrates the design space for robust and transparent path-vector policy systems. (This figure is meant to aid in developing intuitions, and should not be taken too literally.) The x-axis represents the expressive power of systems, and the y-axis represents the relative difficulty of checking the global constraint. Combinations of path-vector systems and policy languages which fall close to the bottom right of Figure 1 are generally desirable.

Some points in the space deserve attention. On the bottom horizontal line lie systems that require no global constraint to be robust. In this paper, we assume "minimal" expressiveness is "Shortest-Paths" routing; a simple extension to this is "Shortest-Available Paths," which

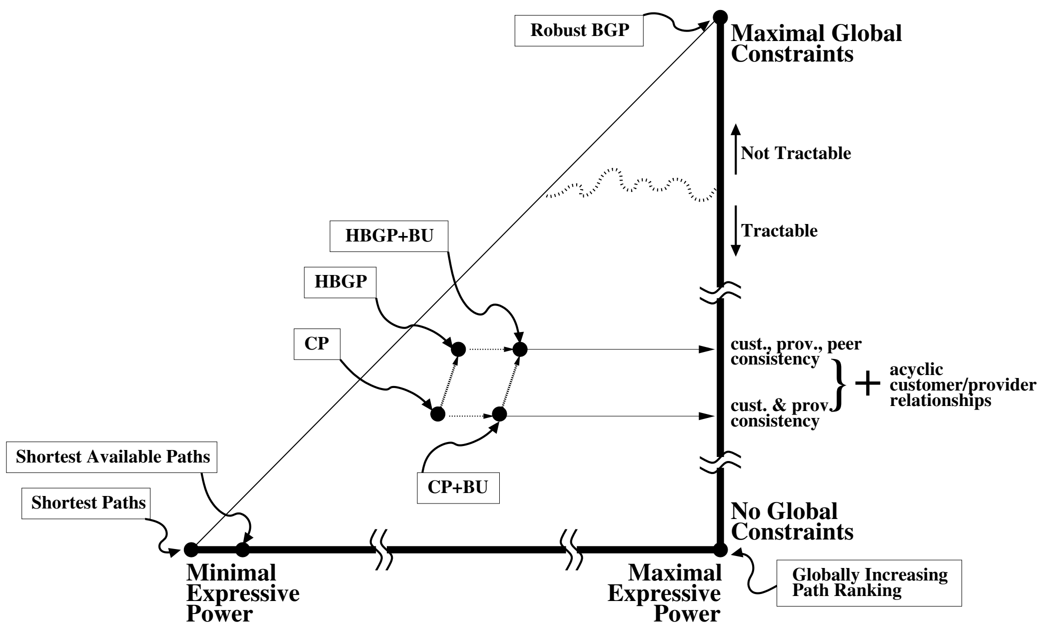

Figure 1: Design space for **robust** and **transparent** path-vector systems.

allows routes to be filtered (even if they are the shortest) and chooses the shortest path from the remaining routes. (Both examples are given in Section 3.) We take "maximal" expressiveness to be the expressive power of a natural class of robust systems that we define in Section 5.3. Two possible systems which possess the property "Globally Increasing Path Ranking" are discussed in Section 6.3; while these achieve maximal expressiveness with no global constraint, they sacrifice other design goals in the process. The final extreme point, "Robust BGP," is a system in which all BGP policies are collected and verified not to contain conflicting policies. One might use the Routing Policy Specification Language (RPSL) [1] in the manner suggested in [7] to accomplish this. Many practical issues make this scenario unlikely; furthermore, it was shown in [10] that, in the worst case, checking various global-consistency constraints is *NP-hard*.

Hierarchical BGP systems (inspired by [5, 6]) provide examples from today's commercial Internet. Figure 1 includes the system CP, a BGP-like system in which the policy language allows nodes to classify neighbors as customers and providers and to rank routes consistent with those relationships; CP is robust if there are no cycles in the customer/provider graph and if classifications of neighbors are consistent. We might increase the expressiveness of this system in two ways: (1) allow an additional classification of neighbors as peers, in which case we must modify the global constraint to additionally check the consistency of peer classifications (the system HBGP); or (2) modify the policy language to permit marking routes for backup use (the system CP+BU). Combining both approaches achieves the expressiveness of the system HBGP+BU. These types of systems are discussed in Section 8. Note that in the real world, there are no existing methods to enforce either the local or global constraints, although Internet economics seems to ensure that networks behave in close approximation to the rules described by the above-mentioned robustness conditions.

2 Path-Vector Policy Systems

In this section, we define the "protocol part" of our framework: the underlying exchange system for route information. We sketch the components independent of any particular system or instance of a system. Using the definitions presented here, we can rigorously explore the protocol design space in later sections.

2.1 Dynamics of Path-Vector Routing

We first briefly discuss the intended dynamics of routing using a path-vector system, as this motivates the system components we define in our framework.

Informally, let each node in the network be a protocol-speaking router responsible for its autonomous domain. A node advertises destinations in its network to its neighbors, and they further transmit this information to their neighbors, *etc*. Whenever a router gets new information about a destination, it determines the best route to that destination given all the up-to-date information it has collected. We expect that routers will influence these decisions by modifying route attributes. This can be done on *export*, when routes are advertised to neighbors (or possibly filtered out altogether), or on *import*, when data are collected and stored for decision-making.

Therefore, we assume that there is some data structure to store and exchange route information, and that transformations to these data structures are made on import and export as dictated by routers' policy configurations. The exchange of these data structures between neighbors as described above will eventually permeate the network with knowledge about the various destinations *originated* by routers. Comparing these data structures gives a "best" route to a destination.

2.2 Formal Definition of Path-Vector Systems

As we develop our framework, we will use a simplified model of BGP as a running example. This example model assumes that each node (router) represents an entire autonomous system and thus treats only External BGP (not Internal BGP). It also ignores most BGP attributes and simplifies others. We will adorn the elements of this example system with the subscript μbgp.

2.2.1 Route Information

A path descriptor is a data record about a path that contains enough information (*e.g.*, the routing destination, the sequence of AS numbers along the entire path, routers' preference values for the path, transmission cost, *etc.*) for a router to compare it to other paths and to inform its neighbors about the path so that they can do the same. A router learns of paths by receiving descriptors from neighbors and preserves knowledge of potential best routes by storing descriptors for paths to all known destinations.

The path-vector-system specification includes a description of the components in a path descriptor and a map that ranks them using values from a totally ordered set. This ranking permits routers to determine best routes based on just the information contained in the available descriptors to a destination; in particular, the rank of a descriptor depends only on that descriptor. Determining rank normally involves some components of path descriptors that can be transformed by both locally configured policies and the underlying message-exchange protocol itself.

Definition 2.1. Let the quadruple

$$\mathcal{I} = (\mathcal{D},\ \mathcal{R},\ \mathcal{U},\ \omega)$$

be the *route-information* portion of the path-vector-system specification. The components are defined as follows:

$\mathcal{D}$ is the set of possible routing destinations;

$\mathcal{R}$ is the set of path descriptors, such that to every $r \in \mathcal{R}$ there must be associated a unique $dest(r) \in \mathcal{D}$;

$\mathcal{U}$ is a set totally ordered by $\leq$; and

ω is a function (the *ranking function*) from $\mathcal{R}$ to $\mathcal{U}$ that determines how path descriptors are ranked (thus, the role of path-descriptor attributes in choosing routes).

Remark 2.2. Although the mechanics of determining "best" routes will be discussed in Section 2.6, we observe the convention that the ranking function will map more preferred paths to smaller elements of $\mathcal{U}$.

Running Example, Part 1. In our example system, let $\mathcal{D}$ be the set of all IPv4 CIDR blocks. Let the set of path descriptors be

$$\mathcal{R}_{\mu bgp} = \mathcal{D}_{\mu bgp} \times \mathbb{N} \times \mathrm{Seq}(\mathbb{N}) \times \mathbb{N} \times 2^{\mathcal{C}},$$

where $\mathbb{N}$ is the set of natural numbers, $\mathrm{Seq}(\mathbb{N})$ is the set of finite sequences of natural numbers, and $\mathcal{C}$ is the set

$\{\text{red, blue, green}\}$. If $r = (d,\ l,\ P,\ n,\ S) \in \mathcal{R}_{\mu bgp}$, then d is the destination of r, l is the *local preference*, P is the *AS path*, n is the *next hop*, and the elements of S are the *colors* of r. Colors are meant to be a very simple model of BGP communities [3].

Let $\mathcal{U}_{\mu bgp} = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ and $\omega((d,\ l,\ P,\ n,\ S)) = (l,\ |P|,\ n)$, with the ordering $\leq_{\mu bgp}$ on $\mathcal{U}_{\mu bgp}$ given by $(l,\ m,\ n) \leq_{\mu bgp} (l',\ m',\ n')$ if and only if:

$$\begin{aligned}&l > l';\ \text{or}\\&l = l',\ m < m';\ \text{or}\\&l = l',\ m = m',\ n \leq n'.\end{aligned}$$

The combination of $\leq_{\mu bgp}$ and $\omega_{\mu bgp}$ prefers higher local preference, with ties broken by preferring smaller AS-path length and then smaller value of the next hop.

2.2.2 Import and Export Policies

Path-vector systems explicitly include operations for importing routes from neighbors and exporting routes to neighbors. Router operators provide separate *import and export configuration policies* to describe router behavior when exchanging route information, *e.g.*, to change path-descriptor attributes for a route affecting its rank or to filter out routes altogether. The set of node policies across the network would therefore be a component of a specific instance of the path-vector system. On a low level, the import and export policies are per-neighbor functions on path descriptors that transform their components to make preference changes in accordance with local policy. We expect that policies will usually be written in a higher-level policy language, which motivates the policy-language component of design.

A path-vector system includes *local-policy constraints* on what import and export policies are allowed. These limits on the expressiveness of local policies can help guarantee robustness and can help ensure that a protocol achieves its goals; *e.g.*, if policies can only add a positive value to a path-cost attribute that alone determines path rank, the path-vector system implements lowest-cost-path routing.

Formally, let elements of the function space $2^{\mathcal{R}} \to 2^{\mathcal{R}}$ be called *policy functions* (these are functions on sets of path descriptors, thus describing transformations on them). We then define local-policy constraints in the following way.

Definition 2.3. Let the triple

$$\mathcal{C} = (\mathrm{L}^{in},\ \mathrm{L}^{out},\ \mathrm{O})$$

be the *local-constraints* portion of the path-vector-system specification. L^{in} and L^{out} are predicates on import and export policy functions, respectively. If $\mathrm{L}^{in}(f)$ or $\mathrm{L}^{out}(f)$ holds, then f is a *legal* local-policy function. Furthermore, we assume that if either $\mathrm{L}^{in}(f)$ or $\mathrm{L}^{out}(f)$ holds, then f satisfies:

(1) for each $X \subseteq \mathcal{R}$, if $|X| = 1$ then $|f(X)| \leq 1$;

(2) for each $X \subseteq \mathcal{R}$, $f(X) = \bigcup_{r \in X} f(\{r\})$; and

(3) for each $r_1,\ r_2 \in \mathcal{R}$, if $f(\{r_1\}) = \{r_2\}$, then $dest(r_1) = dest(r_2)$.

O is a predicate defined on subsets of $\mathcal{R}$ used to define what sets of path descriptors can be originated at a node. A node can only advertise newly originated destinations described by $X \subseteq \mathcal{R}$ if $\mathrm{O}(X)$ holds.

Running Example, Part 2. In our simplified-BGP example, we want policies to affect only the local-preference and colors (communities) attributes of path descriptors. We let $\mathrm{L}^{in}_{\mu bgp}(f)$ and $\mathrm{L}^{out}_{\mu bgp}(f)$ hold if and only if f satisfies conditions (1)–(3) above as well as

(4) $f((d,\ l,\ P,\ n, S)) = \{(d',\ l',\ P',\ n',\ S')\}$ implies $d' = d$, $P' = P$, and $n' = n$.

Additionally, the only path descriptors which may be originated by nodes are those with an AS path containing the AS alone (because the destination should be in the originating AS's domain) and a default local preference of 0, so we let $\mathrm{O}_{\mu bgp}(X)$ be true if and only if $(d,\ l,\ P,\ n,\ S) \in X$ implies $l = 0$ and $P = v$ where v is the originating AS.

2.2.3 Application of Policies

Although import and export policies allow router operators to configure their routers, we must recognize that it is the router (or the protocol itself) actually applies those policies to path descriptors encountered while running the protocol. Therefore, path-vector-system specifications include a *policy-application function* for both the import and export operations. These functions describe the transformations used by the protocol to apply operator-provided policies to path descriptors. This allows the application of policies to be consistent with the goals of the protocol, *e.g.*, routers may only apply policies when they satisfy a local condition guaranteeing robustness. These functions are often used to make changes to path descriptors uniformly throughout all information exchanges in addition to applying the operator-provided configuration policy (*e.g.*, appending a node name to the described path or hiding certain attributes when they contain private information). Formally, we have

Definition 2.4. Let the pair

$$\mathcal{T} = (t^{in},\ t^{out})$$

be the *protocol-transformation* portion of the path-vector-system specification. Both t^{in} and t^{out} are functions of type $(\mathbb{N} \times \mathbb{N} \times (2^{\mathcal{R}} \to 2^{\mathcal{R}}) \times 2^{\mathcal{R}}) \to 2^{\mathcal{R}}$; the first two arguments are node names, the third is the policy function to apply, the fourth is the target set of path descriptors.

Running Example, Part 3. We now give the protocol transformations for our model of BGP. If u and v are nodes, f is a policy function (expected to be u's export policy function for v), and X is a set of path descriptors (expected to be known to u), then

$$\begin{aligned} t^{out}_{\mu bgp}(u,\ v,\ f,\ X) \ = \{&(d,\ 0,\ vP,\ u,\ S) \\ &\mid (d,\ m,\ P,\ w,\ S) \in f(X)\}. \end{aligned}$$

The protocol applies the (export) policy function (which may change local preference and colors) and then updates the AS-path and next-hop values to reflect the edge $\{u,\ v\}$ in the extended path. It also sets the local preference value to 0, hiding this value from the node receiving information about this path. If Y is a set of path descriptors (expected to be $t^{out}_{\mu bgp}(u,\ v,\ f,\ X)$) and g is v's import policy function for u, then we let

$$\begin{aligned} t^{in}_{\mu bgp}(v,\ u,\ g,\ Y) = \{\, &g(r) \mid r \in Y, \\ &r \text{ describes a simple path}\}. \end{aligned}$$

The protocol thus takes care of filtering any paths which contain loops.

2.2.4 Path-Vector System

Definition 2.5. A *path-vector system* is a triple of the form

$$PV = (\mathcal{I},\ \mathcal{C},\ \mathcal{T})$$

where the components are as defined in Definitions 2.1–2.4.

2.3 Policy Languages

Of course, policy writers don't actually write mathematical functions, but rather write specifications in a *path-vector policy language*. We expect that such languages can be given a rigorous semantics so that policies written in the language can be treated as specifications for functions on path descriptors. A policy language essentially is a local constraint on the policy functions that can be written for a path-vector system. Policy-language designers must ensure that legal policy specifications are guaranteed to have semantics that conform to the constraints of the target path-vector system(s). In practice, this may involve some type of *compilation* to low-level, vendor-specific configuration commands—a transformation that may be rather complex. However, separating the definition of a policy language from the definition of a path-vector system allows us to consider multiple policy languages for the same path-vector system. We can also discuss using different path-vector systems to implement the same policy language.

Definition 2.6. A *policy language* PL for a path-vector system is a language and a *semantic function* $\mathcal{M}$ that maps each *policy configuration* p written in this language to a triple

$$\mathcal{M}(p) = (m^{in},\ m^{out},\ m^{orig})$$

of partial functions of types

$$\begin{aligned} m^{in},\ m^{out} \quad &: \quad V \times V \to (2^{\mathcal{R}} \to 2^{\mathcal{R}}) \\ m^{orig} \quad &: \quad V \to 2^{\mathcal{R}} \end{aligned}$$

If u and v are node identifiers, then $m^{in}(v,\ u)$ and $m^{in}(v,\ u)$ are called the *import* and *export policy functions at* v *for* u, respectively, and $\mathrm{L}^{in}(m^{in}(v,\ u))$ and $\mathrm{L}^{out}(m^{out}(v,\ u))$ hold whenever these policy functions are defined. These functions transform sets of path descriptors. Finally, the function m^{orig} maps node identifiers v to finite subsets of $\mathcal{R}$ such that $\mathrm{O}(m^{orig}(v))$ holds whenever $m^{orig}(v)$ is defined.

We take policy configurations to be the language-specific definitions of policies for one or more nodes; the set of valid policy configurations is part of the language PL.

Running Example, Part 4. We define a simple policy language $PL_{\mu bgp}$. A policy configuration in this language is a list of *declarations*, each having one of the forms:

$$\begin{array}{c} \text{export from } v \text{ to } W \ : \ \textbf{rule} \\ \text{import at } v \text{ from } W \ : \ \textbf{rule} \\ \text{originate from } v \ : \ (d,\ 0,\ \epsilon,\ v,\ S) \end{array}$$

The first and second type declare export and import policies, respectively, and the third type declares routes to be originated from a node. The sets W represent all of the neighboring nodes to which a given declaration is applied. Each **rule** is a transformation of objects in $\mathcal{R}_{\mu bgp}$ defined by a list of *clauses*:

$$\begin{array}{ccc} C_1 & \Longrightarrow & A_1 \\ C_2 & \Longrightarrow & A_2 \\ \vdots & \vdots & \vdots \\ C_n & \Longrightarrow & A_n \end{array}$$

where each C_i is a boolean predicate over path descriptors and each A_i is an *action* to be taken on the input path descriptor. The actions are either of the form **reject**, or they are statements that modify the local preference or colors of a path descriptor. For each path descriptor r input to such a rule, the action associated with the first predicate that evaluates to **true** is performed on r. If no clause matches, the empty set is returned. $\mathcal{M}_{PL_{\mu bgp}}(p)$ is easy to determine given the form of policy configurations in $PL_{\mu bgp}$; see part 5 of the running example in the following subsection.

2.4 Instances of Path-Vector Systems

Definition 2.7. An *instance* of a path-vector system PV with respect to a policy language PL (or *an instance of* $(PV,\ PL)$) is a pair

$$I = (G,\ P),$$

where $G = (V,\ E)$ is an undirected graph, called the *signaling graph*, and the *configuration function* P maps nodes $v \in V$ to policy configurations $P(v) = p_v$ in the policy language PL so that $\mathcal{M}(p_v) = (F_v^{in},\ F_v^{out},\ F_v^{orig})$. We require that $F_v^{orig}(v)$ is defined and that, for every $\{v,\ u\} \in E$, both $F_v^{in}(v,\ u)$ and $F_v^{out}(v,\ u)$ are defined. We will assume that the vertex set V is a subset of $\mathbb{N}$.

Let $F(I) = (F^{in},\ F^{out},\ F^{orig})$ where

$$\begin{array}{rcl} F^{in}(v,\ u) & = & F_v^{in}(v,\ u) \\ F^{out}(v,\ u) & = & F_v^{out}(v,\ u) \\ F^{orig}(v,\ u) & = & \bigcup_{w \in V} F_w^{orig}(v) \end{array}$$

F is a summary configuration function for the instance that represents the collection of policy configurations provided by nodes in the instance. However, F technically describes transformations on path descriptors, and thus is a somewhat “compiled” or “lower-level” version of the policies for the instance, independent of the policy language used to specify them.

Remark 2.8. In most cases, nodes will not originate descriptors on behalf of other nodes, *i.e.*, $F_w^{orig}(v) = \emptyset$ for $w \neq v$, and nodes will not have policies for non-incident edges, *i.e.*, $F_w^{in}(v,\ u), F_w^{out}(v,\ u)$ are not defined for $w \neq v$. In addition, we suggest and often assume that the origination constraint includes a clause to check that nodes only originate path descriptors for destinations they represent or contain, *i.e.*,

$$\mathrm{O}(X) \Rightarrow \left[r \in X \Rightarrow \left(dest(r) = v \Rightarrow r \in F_v^{orig}(v)\right)\right]$$

Definition 2.9. Given two instances $I = (G,\ P)$ and $I' = (G',\ P')$ of $(PV,\ PL)$, the instance I' is said to be a *sub-instance* of I if G' is a subgraph of G and the configuration function P' is equal to P when restricted to G'. For example, given any instance $I = (G,\ P)$ and G', a subgraph of G, the instance $I' = (G',\ P)$ is a sub-instance of I.

Running Example, Part 5. One instance of $(PV_{\mu bgp},\ PL_{\mu bgp})$ consists of the five-vertex graph shown in Figure 2 and policy configurations in Figure 3.

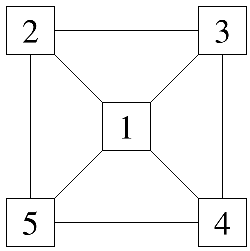

Figure 2: A simple 5 node graph.

2.5 Realizable Path Descriptors

We are particularly interested in the path descriptors that arise as the result of first originating a path descriptor at some node and then forwarding it along some path in the network, applying the appropriate export, import, and protocol transform functions along the way. We call these *realizable path descriptors*. Because we do not usually make use of the path descriptors that arise after applying an export transform but before applying the corresponding input transform, we combine these functions into *arc policy functions* for convenience.

Definition 2.10. Let I be an instance of $(PL,\ PV)$ with signaling graph $G = (V,\ E)$; let $\{v,\ u\} \in E$ be any edge. Then the *arc policy function* $F_{(v,u)}$ is the function which takes the path descriptors at u and produces the path descriptors that v has after import from u. Thus, for $X \subseteq \mathcal{R}$,

$$\begin{aligned} F_{(v,u)}(X) = t^{in}(\,&v,\ u,\ F^{in}(v,u), \\ &t^{out}(u,\ v,\ F^{out}(u,v),\ X)). \end{aligned}$$

Note that it may be the case that $F_{(v,u)}(X) = \emptyset$ for some $X \neq \emptyset$. In this case we say that the path descriptors of X have been *filtered out by* $F_{(v,u)}$.

Conditions (1)–(3) given in Definition 2.3 only need to hold for the functions $\{F_{(v,\ u)} \mid \{v,\ u\} \in E\}$; however, because t^{out} and t^{in} are specified separately from the policies F^{out} and F^{in}, it may be easier for those designing the protocol transformations t^{in} and t^{out} to assume that all policies satisfying L^{out} or L^{in} also satisfy these conditions (and for the compilers of policies into functions to know that it suffices to satisfy these conditions).

originate from 1 : $(d,\ 0,\ (1),\ 1,\ \emptyset)$
export from 1 to $\{2\}$:
  true $\Longrightarrow r$.colors := {red}
export from 1 to $\{3,\ 4\}$:
  true $\Longrightarrow r$.colors := {blue}
export from 1 to 5 :
  true $\Longrightarrow r$.colors := {green}

import at 2 from $\{1,\ 3,\ 5\}$:
  blue $\in r$.colors $\Longrightarrow r$.local-preference := 100
  red $\in r$.colors $\Longrightarrow r$.local-preference := 50
  green $\in r$.colors $\Longrightarrow r$.local-preference := 10
export from 2 to $\{3,\ 5\}$:
  true $\Longrightarrow r$

import at 3 from $\{1\}$:
  true $\Longrightarrow r$.local-preference := 100
import at 3 from $\{2,\ 4\}$:
  green $\in r$.colors $\Longrightarrow r$.local-preference := 1000
  blue $\in r$.colors $\Longrightarrow r$.local-preference := 500
export from 3 to $\{2,\ 4\}$:
  true $\Longrightarrow r$

import at 4 from $\{1\}$:
  true $\Longrightarrow r$.local-preference := 10
import at 4 from $\{3,\ 5\}$:
  green $\in r$.colors $\Longrightarrow r$.local-preference := 50
  blue $\in r$.colors $\Longrightarrow r$.local-preference := 25
export from 4 to $\{3,\ 5\}$:
  true $\Longrightarrow r$

import at 5 from $\{1,\ 2,\ 4\}$:
  green $\in r$.colors $\Longrightarrow r$.local-preference := 2
  red $\in r$.colors $\Longrightarrow r$.local-preference := 1
export from 5 to $\{2,\ 4\}$:
  true $\Longrightarrow r$

Figure 3: Example policy configurations in $PL_{\mu bgp}$.

Suppose that the path P is a simple path in G from a node v to node w; we write this as a sequence $P = vx_1 \dots x_k w$ of distinct nodes starting with v and ending with w. If $r_w \in F^{orig}(w)$, then we let $r(P,\ r_w) \subseteq \mathcal{R}$ be the result of passing r_w along P and applying the corresponding arc policies. Formally, if $P = w$, set $r(w, r_w) = \{r_w\}$. If $v \neq w$ then write $P = vx_1 \dots x_k w = vP'$ and let $r(vP', r_w) = F_{(v,x_1)}(r(P', r_w))$.

Definition 2.11. The set of path descriptors *realizable at* u *in* I is the set $\mathcal{R}_I^u$ of descriptors which may be originated at u or which may be obtained by (legally) originating a descriptor elsewhere and passing it along a network path, successively transforming it with the appropriate arc policies. Formally:

$$\begin{aligned}\mathcal{R}_I^u = \; & F^{orig}(u) \cup \\ & \{r' \in r(P,\ r_w) \mid w \in V,\ r_w \in F^{orig}(w), \\ & \text{and } P \text{ is a path from } u \text{ to } w\}.\end{aligned}$$

2.6 Path-Vector Solutions

A solution for an instance of a path-vector system is an assignment of path descriptors to nodes which is both realizable and which satisfies each node's preferences to as great an extent as possible given the assignments to the surrounding nodes.

Definition 2.12. A *path assignment* ρ is a mapping from V to $2^{\mathcal{R}}$. Given a path assignment ρ, define the set $C(\rho,\ v)$ of *candidates* at node v to be

$$F^{orig}(v) \cup \{r \in \mathcal{R} \mid \{v,\ u\} \in E \wedge\ r \in F_{(v,\ u)}(\rho(u))\},$$

i.e., those path descriptors which are either originated at v or which are the result of importing descriptors assigned by ρ to v's neighbors.

Definition 2.13. For $X \subseteq \mathcal{R}$, let the set $\min(X)$ be the set of descriptors in X (for all destinations) which are minimally ranked among the descriptors with the same destination, *i.e.*, define $\min(X) =$

$$\{r \in X \mid \forall r'\ dest(r') = dest(r) \Rightarrow\ \omega(r) \leq \omega(r')\}.$$

The assignment ρ is a *solution for* I if for each $v \in V$ we have (1) $\rho(v) \subseteq \mathcal{R}_I^v$ and (2) $\rho(v) = \min(C(\rho,\ v))$.

For the instance I, let $sol(I)$ be set of solutions for I. Note that it may be the case that $sol(I) = \emptyset$.

Running Example, Part 6. The unique solution $\rho_{\mu bgp}$ to the instance from part 5 of our running example is shown in Table 1. Note that the sub-instance obtained by deleting the edge $\{1,\ 5\}$ from the graph has two solutions; so, this instance is not robust.

3 Examples

We first discuss two points in the design space that were mentioned in the overview and then present an additional, more complex example.

v	$\rho_{\mu bgp}(v)$
1	$\{(d,\ 0, (1),\ 1,\ \emptyset)\}$
2	$\{(d,\ 50,\ (2, 1),\ 1,\ \{red\})\}$
3	$\{(d,\ 1000,\ (3, 4, 5, 1),\ 4,\ \{green\})\}$
4	$\{(d,\ 50,\ (4, 5, 1),\ 5,\ \{green\})\}$
5	$\{(d,\ 2,\ (5, 1),\ 1,\ \{green\})\}$

Table 1: Unique solution for our running example.

3.1 Shortest-Paths Routing

Example 3.1. (Shortest Paths) Let $\mathcal{R}_{sp} = \mathcal{D}_{sp} \times \mathbb{N} \times \mathrm{Seq}(\mathbb{N})$. The second component of $r \in \mathcal{R}_{sp}$ is a non-negative length associated with the path in the third component of r; this length is the sole factor in path ranking, with shorter paths preferred. We permit nodes to increment the length of a path on import or export, so that $\mathrm{L}^{in} = \mathrm{L}^{out} = \mathrm{L}_{sp}$ where $\mathrm{L}_{sp}(f)$ holds iff there exists a positive integer n such that for all $d \in \mathcal{D}_{sp},\ m \in N,\ P \in$ Seq($\mathbb{N}$), we have $f(\{(d,\ m,\ P)\}) = \{(d,\ m + n,\ P)\}$.

We define the export policy-application function $t_{sp}^{out}(u,\ v,\ f,\ X)$ to produce the set

$$\{(d,\ m, uP) \mid (d,\ m,\ P) \in f(X)\}.$$

That is, t_{sp}^{out} merely extends the path P with the node u. We define the import policy-application function $t_{sp}^{in}(u,\ v,\ f,\ X)$ to produce the set

$$f(\{r \mid r = (d,\ l, P) \in X \text{ where } P \text{ is a simple path}\}).$$

That is t_{sp}^{in} eliminates path descriptors with a loop, and then applies the import policy.

Remark 3.2. Note that by replacing Seq($\mathbb{N}$) with $\mathbb{N}$ we could model "distance vector" protocols similar to RIP [13]. However, we will restrict our attention to those systems that do not allow signaling paths of arbitrary length.

Example 3.3. (Shortest-Available Paths) This system is a slight extension of Shortest Paths in which path descriptors can be filtered out, both on import and export. We simply modify the local constraints L^{in} and L^{out} to allow filtering, leaving all other definitions unchanged. The new constraint $\mathrm{L}_{sap}(f)$ holds iff there exists a positive integer n such that for all $d \in \mathcal{D}_{sp},\ m \in N,\ P \in$ Seq($\mathbb{N}$), either $f(\{(d,\ m,\ P)\}) = \emptyset$ or $f(\{(d,\ m,\ P)\}) = \{(d,\ m + n,\ P)\}$.

3.2 A Catalan Example

We now give an example which is rather unlike traditional routing problems and which suggests the broad applicability of the framework we have presented. The policy-application functions of this path-vector system ensure that the path descriptors which are passed between nodes are those whose paths are subpaths of lattice paths related to the famous Catalan numbers. We thus denote this path vector system by PV_{cat}. The set $\mathcal{U}_{cat}$ includes ∞, and the ranking function ω_{cat} is constructed so that exactly the desired lattice paths are given finite rank; subpaths of the desired paths are not filtered but instead given infinite rank.

The policies written by nodes in an instance of this system do not affect which paths are imported and exported; they only determine the rank of the path descriptors which are constrained by PV_{cat} to have finite rank. Given the myriad of combinatorial interpretations of the Catalan numbers, there are many ways that nodes in an instance of PV_{cat} can interpret and then "naturally" order the path descriptors that they receive from their neighbors. We suggest a few such policies below.

3.2.1 The Path Vector System PV_{cat}

We assume that each node in an instance of PV_{cat} has a neighbor one step to the north and one step to the east (as though points with integer coordinates in $\mathbb{R}^2$) and that the protocol knows the spatial relationship between neighbors.

Let $\mathrm{Seq}(0,1)$ be the set of all finite 0–1 sequences, and let $\mathcal{R}_{cat} = \mathcal{D}_{cat} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathrm{Seq}(0,1)$. We then make the following definitions.

$$\begin{aligned} \mathcal{U}_{cat} &= \mathbb{N} \cup \{\infty\} \\ dest_{cat}(d,\ x,\ y,\ z,\ P) &= d \\ \omega_{cat}(d,\ x,\ y,\ z,\ P) &= \begin{cases} z, & x = y \\ \infty, & \text{otherwise} \end{cases} \end{aligned}$$

In $r = \{d,\ x,\ y,\ z,\ P\}$, we will use P to encode the corresponding path (using 0 for east steps and 1 for north steps) and x and y to store the number of east and north steps in the path.

We let $\mathrm{O}_{cat}(X)$ hold if and only if $r \in X \Rightarrow r = (d,\ 0,\ 0,\ m,\ \epsilon)$, where ϵ is the empty sequence. Let $\mathrm{L}_{cat}(f)$ hold if and only if for every $r = (d,\ x,\ y,\ z,\ P) \in \mathcal{R}_{cat}$, $f(\{r\}) = \{(d,\ x,\ y,\ z',\ P)\}$, so that f may only change the fourth element of the path descriptor. We take L_{cat} to be the constraint on both import and export functions.

$$\begin{aligned} \mathrm{L}^{in}_{cat}(f) &= \mathrm{L}_{cat}(f) \\ \mathrm{L}^{out}_{cat}(f) &= \mathrm{L}_{cat}(f) \end{aligned}$$

Remark 3.4. Note that L_{cat} ensures that policies do not filter paths as in Shortest Paths (Example 3.1). This could be changed to allow filtering as in Shortest-Available Paths (Example 3.3).

We define the export policy-application function $t^{out}_{cat}(u,\ v,\ f,\ X)$ to be the set

$$\{(d,\ x+1,\ y,\ z,\ 0P) \mid (d,\ x,\ y,\ z,\ P) \in f(X)\}$$

if v is 1 step east of u, the set

$$\{(d,\ x,\ y+1,\ z,\ 1P) \mid (d,\ x,\ y,\ z,\ P) \in f(X)\}$$

if v is 1 step north of u, and $\emptyset$ otherwise. Thus t^{out}_{cat} restricts the export of descriptors to those neighbors which are one step east or north from the exporting node. It also updates the path P, prepending a 0 or 1, depending on whether this export is to the east or north, and the total number of east (x) and north (y) steps in P. Note that we do not make assumptions about the labels of the nodes (although we could express these restrictions using node labels from $\mathbb{N} \times \mathbb{N}$). We define $t^{in}_{cat}(u,\ v,\ f,\ X)$ to be the set

$$f(\{(d,\ x,\ y,\ z,\ P) \in X \mid y \geq x\}).$$

The combination of t^{out}_{cat} and t^{in}_{cat} ensures that the path descriptors which have not been filtered correspond to paths with north and east steps and that, starting at the destination, have never made more east steps than north steps as they are forwarded. It is well known that the number of such paths with exactly n steps north and n steps east is the n^{th} Catalan number $\frac{1}{n+1}\binom{2n}{n}$. The definition of ω_{cat} means that the path descriptors which have finite rank are exactly those which have passed along equally many north and east steps. While PV_{cat} determines the set of descriptors which are assigned finite rank at each node, it has no impact on the ordering of the descriptors in this set. These rankings will be determined by the policies of nodes in an instance of PV_{cat} and may correspond to natural orderings on some of the many families of objects counted by the Catalan numbers (66 examples of which are given in Exercise 6.19 of [24]).

3.2.2 Policies for PV_{cat}

Assume that we have some policy language PL_{cat} for PV_{cat} in which a node can describe: a family of objects counted by the Catalan numbers; a ranking of these objects; and an appropriate bijection between the objects and Catalan sequences. (A *Catalan sequence* of size n is an element of $\mathrm{Seq}(0, 1)$ with n 0s and n 1s, such that no initial subsequence has more 0s than 1s.) We now consider different policy functions, compiled from policies written in PL_{cat} and which satisfy L_{cat}, which may arise in PV_{cat}. These functions must be of the form

$$f(\{d,\ x,\ y,\ z,\ P\}) = \{(d,\ x,\ y,\ z',\ P)\},$$

so we will define the functions below by defining z' in each instance.

The first two examples use as objects lattice paths (*i.e.*, composed of the steps $(1, 0)$ and $(0, 1)$) from $(0, 0)$ to (n, n) which never fall below the diagonal $y = x$. They also use the bijection described in the definition of PV_{cat} in which a 1 appearing in an element of $\mathrm{Seq}(0, 1)$ corresponds to a step of $(0, 1)$ in a lattice path. For the first example, let the ranking of a path be its *area*, *i.e.*, the number of whole squares below the path and above the diagonal $y = x$. The import function then sets z' to be the area of the path corresponding to P. For our second example, we prefer shorter paths to longer ones, and given two paths of the same length, we prefer the one which has the $(1, 0)$ step at the first step where they differ. For a sequence P of length $2n$, the import function then sets z' to be $\sum_{i=1}^{n-1} \binom{2i}{i}/(i + 1)$ plus the number of paths of length $2n$ that have a $(1, 0)$ step in the first place where they differ from P.

Among all paths of length $2n$, the path along the diagonal (alternating north and east steps) will be the most preferred using both of these policies, while the path consisting of n steps north followed by n steps east will be the least preferred. However, the first policy will prefer *any* path along the diagonal to any other path, regardless of the lengths of the two paths, in contrast to the second policy. They will also disagree on the relative rankings of the two paths encoded by $P_1 = 1011111\ldots00000\ldots$ and $P_2 = 110010101010\ldots$.

Policies might also be written which view the object encoded by a sequence P of length $2n$ as an ordering π of $\{1, \ldots, n\}$ which does not have three (possibly non-adjacent) elements in decreasing order (a 321-avoiding permutation). (See [24] for a bijection to the lattice paths we have been considering.) The import function could assign to z' any number of values, including various permutation statistics (*e.g.*, descents, inversions) evaluated on π. Once the path P is viewed as a permutation, there are a wide variety of ways to define z.

4 Expressiveness

To rigorously capture the *expressive power* of path-vector systems, we use a variant of the Stable Paths Problem (SPP) [9] as a sematic domain. After reviewing the SPP framework, we show how to map path-vector instances to equivalence classes of SPP instances and use this to compare the expressiveness of path-vector policy systems.

4.1 The Stable Paths Problem (SPP)

Definition 4.1. The quadruple

$$S = (G,\ v_0,\ \mathcal{P},\ \Lambda)$$

is an *instance of the Stable Paths Problem (SPP)* if $G = (V,\ E)$ is a finite undirected graph, $v_0 \in V$ (called the *origin*), $\mathcal{P}$ is a set of simple paths in G terminating at v_0, and the mapping Λ takes nodes $v \in V$ to a path ranking function $\lambda^v = \Lambda(v)$. Each λ^v is a function that takes a path in $\mathcal{P}^v = \{P \in \mathcal{P} \mid P \text{ is a path starting at } v\}$ to its *rank* in $\mathbb{N}$. If $W \subseteq \mathcal{P}^v$, then the subset of "best paths" in W, $\min(\lambda^v,\ W) \subseteq W$, is defined as the set

$$\{P \in W \mid \text{for every } P' \in W,\ \lambda^v(P) \le \lambda^v(P')\}.$$

Definition 4.2. A *path assignment* for an SPP-instance S is any mapping π from V to subsets of $\mathcal{P}$ such that $\pi(v) \subseteq \mathcal{P}^v$. The set $\mathrm{candidates}(u, \pi)$ consists of all permitted paths at u that can be formed by extending the paths assigned to neighbors of u. For $u = v_0$, $\mathrm{candidates}(u, \pi) =$

$\{(u)\}$, and for $u \neq v_0$, candidates(u, π) equals

$$\{uQ \in \mathcal{P}^u \mid \{v,\ u\} \in E \text{ and } Q = \pi(v)\}.$$

A path assignment π is a *solution* for an SPP if for every node u we have $\pi(u) = \min(\lambda^u,\ \text{candidates}(u, \pi))$. That is, if F is a functional that takes path assignments π to path assignments $F(\pi)$, defined as $F(\pi)(u) = \min(\lambda^u,\ \text{candidates}(u, \pi))$, then the solutions of the SPP are exactly the fixed points of F (for any solution π we have $F(\pi) = \pi$, and $F(\pi) = \pi$ implies π is a solution).

A convenient abbreviation for the best path at u under π is defined to be best$(u, \pi) = \min(\lambda^u,\ \text{candidates}(u, \pi))$. Then π is a solution if $\pi(u) = \text{best}(u, \pi)$ at each node u.

Remark 4.3. The definition for SPP given here is a bit more general than that of [9] in that we do not require "strictness," which guarantees that $|\pi(v)| \leq 1$ for every solution π. In addition, we have changed the order of the ranking to prefer paths with smaller (not larger) rank. Finally, we have allowed any node $v_0 \in V$ to be the origin.

4.2 Mapping Path-Vector Systems to SPP Instances

Suppose that $I = (G(V, E),\ F)$ is an instance of some $(PV,\ PL)$. We may represent I as a set of instances of the Stable Paths Problem (SPP). For each $w \in V$ and each $r_w \in F^{orig}(w)$ we construct an SPP instance $S_{(I,w,r_w)}$.

Definition 4.4. Define $I(w,\ r_w)$ to be a *restriction* of instance I where the only descriptor originated is r_w at node w. Given $I(w,\ r_w)$, define the corresponding SPP instance $S_{(I,w,r_w)}$ as described below, and let $\mathcal{S}(I) = \{I(w,\ r_w) \mid w \in V, r_w \in F^{orig}(w)\}$ be the set of all SPP instances which correspond to a restriction of I.

Let the set of permitted paths in $S_{(I,w,r_w)}$ be $\mathcal{P}_{(I,w,r_w)} = \{P \mid r(P,\ r_w) \neq \emptyset\}$. For each $v \in V$, set the values of the ranking function $\lambda^v_{(I,w,r_w)}$ such that the following holds: $\lambda^v_{(I,w,r_w)}(P_1) \leq \lambda^v_{(I,w,r_w)}(P_2)$ if and only if $\{r_1\} = r(P_1,\ r_w)$, $\{r_2\} = r(P_2,\ r_w)$, and $\omega(r_1) \leq \omega(r_2)$.

It may be that $\lambda^v_{(I,w,r_w)}(P_1) = \lambda^v_{(I,w,r_w)}(P_2)$ for paths $P_1 \neq P_2$. This can happen in one of two ways. First, it may be the case that $r(P_1,\ r_w) = r(P_2,\ r_w)$. That is, two distinct signaling paths may result in the same path descriptor. Or, it may be the case that $r_1 = r(P_1,\ r_w) \neq r(P_2,\ r_w) = r_2$, but $\omega(r_1) = \omega(r_2)$.

There is an exact correspondence between the set of solutions for I and the set of solutions for $\mathcal{S}(I)$ as shown by the following theorems. (The proofs are mostly algebraic manipulation using the definitions above and we defer them to Appendix A.)

Theorem 4.5. *If π is a solution for $S_{(I,w,r_w)}$, then*

$$\rho_\pi(v) = \bigcup_{P \in \pi(v)} r(P,\ r_w)$$

is a solution for $I(w,\ r_w)$.

Theorem 4.6. *If ρ is a solution for $I(w,\ r_w)$, then*

$$\pi_\rho(v) = \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \rho(v)\}$$

is a solution for $S_{(I,w,r_w)}$.

Theorem 4.7. *$\pi_{\rho_\pi} = \pi$ and $\rho_{\pi_\rho} = \rho$.*

Running Example, Part 7. An SPP corresponding to our running example is presented in Figure 4. Node 1 is the origin. Next to each node are the permitted paths of that node listed in order of preference, starting with the most preferred at the top. Note that the actual values of the ranking function are not important, only the relative preference of each permitted path at each node; this figure can be taken to represent an entire equivalence class of SPPs with different values for each λ^v but the same orderings on each set $\mathcal{P}^v$.

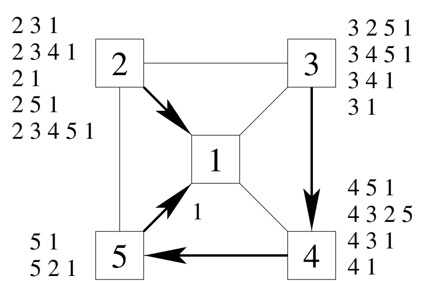

Figure 4: SPP for running example.

4.3 Definition of Expressive Power

Two distinct SPPs can represent the same set of solutions because the specific values in $\mathbb{N}$ that a ranking function λ^v takes on are not really important—what is important is the *relationship* between the rankings of permitted paths at a given node v.

For any SPP instance S, define two relations, $\ominus_S$ and $\oslash_S$, on permitted paths $\mathcal{P}$. First, $P_1 \ominus_S P_2$ if and only if $P_1,\ P_2 \in \mathcal{P}$ and P_1 is a subpath of P_2, *i.e.*, there exists a path Q (possibly ϵ, the empty path) such that $QP_1 = P_2$. Note that $\ominus_S$ is a partial order on permitted paths. Second, $P_1 \oslash_S P_2$ if and only if there is a $v \in V$ such that $P_1,\ P_2 \in \mathcal{P}^v$ and $\lambda^v(P_1) \leq \lambda^v(P_2)$. Define relation $\odot_S$ to be the transitive closure of the relation $\ominus_S \cup \oslash_S$.

Definition 4.8. We say that two SPPs S_1 and S_2 are *equivalent* if they are defined on the same graph, have the same set of permitted paths, and $\odot_{S_1} = \odot_{S_2}$. Define the set $\mathcal{E}(S)$ to be the set of all SPPs equivalent to S.

Definition 4.9. We define the *expressive power* of a path-vector policy system $(PV,\ PL)$ as the set $\mathcal{M}(PV,\ PL) =$

$$\{\mathcal{E}(S) \mid S \in \mathcal{S}(I) \text{ for some } (PV,\ PL) \text{ instance } I\}.$$

$\mathcal{M}(PV)$ means the maximal expressive power of PV when it is not constrained by a policy language, *i.e.*, the maximal expressive power of PV with respect to a policy language allowing all legal policy functions to be expressed.

Remark 4.10. We note that

$$\mathcal{M}(PV_{sp}) \subsetneq \mathcal{M}(PV_{sap}) \subsetneq \mathcal{M}(PV_{\mu bgp}).$$

Shortest-Available Paths (PV_{sap}) allows nodes to filter routes while Shortest Paths (PV_{sp}) does not. Any routing configuration in PV_{sp} is captured by PV_{sap}. But, given any configuration permitted in PV_{sp}, we can filter one of the routes and obtain a new configuration where the policies are permitted by PV_{sap} but not PV_{sp}; thus, $\mathcal{M}(PV_{sp}) \subsetneq \mathcal{M}(PV_{sap})$. Likewise, because $PV_{\mu bgp}$ essentially allows nodes to rank routes in any order, it permits a routing configuration where a node prefers a longer path to a shorter one. Therefore its expressive power is more than that of PV_{sap}.

5 Robustness

We first define robustness using SPP semantics and then present a natural class of expressive, robust SPPs, characterizing this class in the path-vector framework.

5.1 Definition of Robustness

Definition 5.1. An instance I over $(PV,\ PL)$ is said to be *robust* if it has a unique solution and every sub-instance of I has a unique solution. If every instance of a path-vector policy system $(PV,\ PL)$ is robust, then $(PV,\ PL)$ is said to be *robust*.

Definition 5.2. In a similar manner, we can define robustness of SPP instances. Define the set $\mathcal{RSPP}$ as

$$\mathcal{RSPP} = \{\mathcal{E}(S) \mid S \text{ is a robust SPP instance}\}.$$

Given the results of the previous section, we then see that a path-vector policy system $(PV,\ PL)$ is robust if and only if

$$\mathcal{M}(PV,\ PL) \subseteq \mathcal{RSPP}.$$

We are interested in the design space of robust path-vector policy systems.

Conjecture 5.3. *For every $(PV,\ PL)$, if $\mathcal{M}(PV,\ PL) \subseteq \mathcal{RSPP}$, then there exists an $\mathcal{E}(S) \in \mathcal{RSPP}$ such that $\mathcal{E}(S) \notin \mathcal{M}(PV,\ PL)$. In other words, no path-vector policy system can capture exactly all robust systems.*

5.2 A Natural Set of Robust Systems

Definition 5.4. The SPP S is *almost-partially ordered* if $\odot_S$ is reflexive, transitive, and obeys the following rule:

Rule 5.5. $P_1 \odot_S P_2$ and $P_2 \odot_S P_1$ implies that $P_1 = P_2$ or $\exists\, v$ such that $P_1,\ P_2 \in \mathcal{P}^v$.

(Traditional notions of antisymmetry and partial ordering for $\odot_S$ do not allow permitted paths of equal rank at any node; thus, we use the slightly modified notion given above.) Then let

$$\mathcal{APOSPP} = \{\mathcal{E}(S) \mid S \text{ is almost-partially ordered}\}$$

be the set of all almost-partially ordered equivalence classes of SPPs.

If the SPP S is almost-partially ordered, then we will write $P_1 \leq P_2$ for $P_1 \odot_S P_2$, and we will write $P_1 < P_2$ if $P_1 \odot_S P_2$ but $P_2 \not\odot_S P_1$.

Theorem 5.6. *If an SPP instance S is almost-partially ordered, then it is robust.*

In order to prove Theorem 5.6, we must introduce another definition from the SPP framework [9].

Definition 5.7. A *dispute wheel* is a cycle of nodes $v_1, v_2, \ldots, v_k, v_{k+1} = v_1$ in an SPP instance such that there exist paths $R_1, R_2, \ldots, R_k, R_{k+1} = R_1$ and $Q_1, Q_2, \ldots, Q_k, Q_{k+1} = Q_1$ such that $Q_i \in \mathcal{P}^{v_i}$, $R_iQ_{i+1} \in \mathcal{P}^{v_i}$, and $\lambda^{v_i}(R_iQ_{i+1}) < \lambda^{v_i}(Q_i)$. The nodes and paths R_i are on the *rim* of the dispute wheel, while the paths Q_i are called the *spokes* of the wheel.

The following lemma connecting dispute wheels and Definition 5.4 will be useful in proving Theorem 5.6.

Lemma 5.8. *The SPP S is almost-partially ordered if and only if it has no dispute wheel.*

Proof. First, suppose that S is almost-partially ordered. Furthermore, suppose that S has a dispute wheel with R_i, Q_i as in Definition 5.7 Because $\lambda^{u_i}(Q_{i-1}) \leq \lambda^{u_i}(R_iQ_i)$, we know that $R_iQ_i \leq Q_{i-1}$ because $\leq$ subsumes relation $\oslash_S$. And because Q_i is a subpath of R_iQ_i, we know that $Q_i < R_iQ_i$. Therefore, $Q_i < Q_{i-1}$. Following this chain of inequalities around the dispute wheel yields the contradiction $Q_i < Q_i$. Therefore, S has no dispute wheel.

For the other direction, suppose that S has no dispute wheel and also assume that S is not almost-partially ordered. If S is not almost-partially ordered, then there must exist paths P_1 and P_2 that violate Rule 5.5 because the relation $\odot_S$ is inherently reflexive and transitive; *i.e.*,

$$\begin{aligned} &\exists\, P_1 \neq P_2 \text{ such that} \\ &\quad \text{(i)} \quad P_1 \odot_S P_2, \\ &\quad \text{(ii)} \quad P_2 \odot_S P_1, \text{ and} \\ &\quad \text{(iii)} \quad \forall_{v \in V} : \{P_1, P_2\} \not\subseteq \mathcal{P}^v \end{aligned}$$

Conditions (i) and (ii) imply that there exist sets of paths $\{Y_i\}$ and $\{Z_j\}$, not necessarily distinct, such that

$$P_1 = Y_1 \ominus_S Y_2 \oslash_S \cdots \ominus_S Y_{n-1} \oslash_S Y_n = P_2$$

and

$$P_2 = Z_1 \ominus_S Z_2 \oslash_S \cdots \ominus_S Z_{n-1} \oslash_S Z_n = P_1,$$

respectively. From (iii) we know that it is not the case that $P_1 \oslash_S P_2$ or that $P_2 \oslash_S P_1$; if $P_1 \ominus_S P_2$ and $P_2 \ominus_S P_1$ then $P_1 = P_2$, which is not possible if P_1 and P_2 violate Rule 5.5. Therefore, there must be intervening distinct paths in the cycle of relationships above, *i.e.*, $(\{Y_i\} \cup \{Z_j\}) \backslash \{P_1, P_2\} \neq \emptyset$. Using the "cycle of paths" in $\{Y_i\} \cup \{Z_j\}$, we can build a dispute wheel: if $X_1 \ominus_S X_2$ for $X_1, X_2 \in \{Y_i\} \cup \{Z_j\}$, then X_1 is a subpath of X_2 and X_1 can be a spoke path while X_2 can be the spoke path X_1 exported to a rim neighbor; then $X_2 \oslash_S X_3$ and X_2 is the rim path preferred to the spoke path X_3, *etc*.

The existence of a dispute wheel in S is a contradiction; thus S is almost-partially ordered. □

With Lemma 5.8 in hand and a result from [9], we can proceed with the proof of Theorem 5.6.

Proof. If S is almost-partially ordered, then by Lemma 5.8 it has no dispute wheel. Then by Theorem V.10 in [9], S is robust. (In particular, Theorem V.3 in [9] states that a dispute-wheel-free S has a solution, Theorem V.4 states that it has a unique solution, Theorem V.9 guarantees that the SPVP algorithm from [9] will converge to a solution for S, and Theorem V.10 guarantees that a unique solution can be found in the presence of link and node failures.) □

Remark 5.9. An alternative proof may be possible using fixed point theory. As remarked in Definition 4.2, the solutions of the SPP are exactly the fixed points of F, because $F(\pi) = \pi$ implies π is a solution, and for any solution π we have $F(\pi) = \pi$. Perhaps there is some relation that we can impose on the function space of path assignments so that if S is almost partially ordered, then: (1) this relation is partially ordered; (2) F is monotonically increasing; and (3) F is continuous with respect to this order. Then the above proof could dispense with dispute wheels and instead use standard fixed point theorems.

Theorem 5.10. *If $\mathcal{M}(PV, PL) \subseteq \mathcal{APOSPP}$, then the path-vector policy system (PV, PL) is robust.*

Proof. This follows from Theorem 5.6. □

Remark 5.11. The above theorems give the broadest-known sufficient condition for robustness and are consistent with the results in [9].

5.3 Increasing Path-Vector Systems

Definition 5.12. The SPP instance S is *increasing* if

$$\lambda^u(Q) < \lambda^v(vQ)$$

for all edges $\{u,\ v\}$ with path Q permitted at u and path vQ permitted at v. (We are comparing the rankings assigned by *different* nodes; these values have no *a priori* relationship.) Let

$$\mathcal{ISPP} = \{\mathcal{E}(S) \mid S \text{ is increasing}\}$$

be the set of all increasing equivalence classes of SPPs.

Theorem 5.13. $\mathcal{APOSPP} = \mathcal{ISPP}$.

Proof. Clearly $\mathcal{ISPP} \subseteq \mathcal{APOSPP}$ because if the SPP S is increasing, its preferences are already consistent with the subpath relation so that $\odot_S$ is an almost-partial order; so, we only need to show that $\mathcal{APOSPP} \subseteq \mathcal{ISPP}$. If S is an SPP such that $\mathcal{E}(S) \in \mathcal{APOSPP}$, then we can *topologically sort* the permitted paths of S. (See Appendix B for details of this process.) We can then create a new SPP S' by creating a new ranking function λ' which both respects this topological order (so that the system is increasing) and which has the same relative preferences as λ. Clearly $\mathcal{E}(S) = \mathcal{E}(S')$; as S' is increasing, $\mathcal{E}(S) \in \mathcal{ISPP}$. ☐

Ideally, we would like to construct a $(PV,\ PL)$ pair such that $\mathcal{M}(PV,\ PL) = \mathcal{ISPP}$, thus obtaining expressiveness and robustness. We now examine two ways to modify the running-example system $PV_{\mu bgp}$ so that the result is an increasing path-vector system. As we see in the next section, each of these systems lacks some desirable property, a conflict which is in fact unavoidable (Theorem 6.9).

Example 5.14. System PV_{up} shares local preferences between nodes (therefore, it is not policy-opaque) and has local policy constraints that enforce increasing rank between neighbors. Modify the definition of t^{out} so that the local-preference value is passed between neighbors:

$$t^{out}_{up}(u,\ v,\ f,\ X) = \{\, (d,\ m,\ uP,\ u) \mid (d,\ m,\ P,\ x) \in f(X)\}.$$

Let the export constraint be

$$\mathrm{L}^{out}_{up}(f) \Leftrightarrow \forall\, r,\ \omega(\{r\}) \leq \omega(f(\{r\}))$$

and let the import constraint be

$$\mathrm{L}^{in}_{up}(f) \Leftrightarrow \forall\, r,\ \omega(\{r\}) < \omega(f(\{r\})).$$

That is, we constrain the legal policies to be those that increase path rank; in theory, such policies can be written because nodes have access to neighbors' local-preference values.

Example 5.15. System PV_{force} modifies both protocol transformations so that they filter out descriptors whose rank does not increase under the application of the policy function in question. If $r = (d,\ l,\ P,\ n) \in X$, define $h(r) = (d,\ 0,\ P,\ n)$. Then let $t^{in}_{force}(u,\ v,\ f,\ X)$ be the set

$$\{f(\{h(r)\}) \mid r \in X \text{ describes a simple path and } \omega(\{r\}) < \omega(f(\{h(r)\}))\}$$

and let $t^{out}_{force}(u,\ v,\ f,\ X)$ be the set

$$\{(d,\ l,\ uP,\ u) \mid r = (d,\ l,\ P,\ x) \in f(X) \text{ and } \omega(\{r\}) \leq \omega(f(\{r\}))\}.$$

Remark 5.16. $\mathcal{M}(PV_{up}) = \mathcal{M}(PV_{force}) = \mathcal{ISPP}$.

6 Autonomy, Transparency, and Policy Opaqueness

6.1 Autonomy

Network operators often require a high degree of *autonomy* when defining routing policies, *i.e.*, they want wide latitude to write policies that reflect their own interests.

We first define a general notion of autonomy. A collection of predicates on path descriptors, such that exactly one predicate holds for each descriptor in $\mathcal{R}$, induces a partition Π of $\mathcal{R}$. A partial order on these predicates induces a partial order on $\mathcal{R}$. A path-vector policy system is autonomous with respect to $(\Pi, \leq_\Pi)$ if there exists a legal policy that ranks routes consistent with the partial order on Π induced by $\leq_\Pi$.

For example, a policy writer may wish to rank routes solely as a function of the value of one particular attribute of descriptors in the system. If he or she is to do so with full freedom, the system must be autonomous with respect to every partial ordering of the collection of predicates which test the value of that attribute. A system without this autonomy may have local-policy constraints preventing the desired policy configuration.) We can say that the space of ordered partitions given which a path-vector policy system is autonomous represents the *autonomy* of the system, and that *full autonomy* is reached when policy writers can write policies consistent with all possible partitions.

Formally, we have the following.

Definition 6.1. A path-vector system PV is *autonomous with respect to partition* Π *of* $X \subset \mathcal{R}$ iff for any partial order $\leq_\Pi$ on the partition, there exists a legal import policy f (*i.e.*, $\mathrm{L}^{in}(f)$ holds) such that for all $r_i \in \Pi_i, r_j \in \Pi_j$ with $\Pi_i <_\Pi \Pi_j$, there exist $\hat{r}_i, \hat{r}_j \in \mathcal{R}$ such that

$$(f(r_i) = \hat{r}_i \text{ and } f(r_j) = \hat{r}_j) \Rightarrow \omega(\hat{r}_i) < \omega(\hat{r}_j).$$

Useful partition types, as described above, include partitions based on attributes, *e.g.*, "Let $r \in \Pi_i \subset \mathcal{R}$ iff $A(r) = i$" (the index set of the partition is the set of possible attribute values $A(r)$). If PV is autonomous with respect to such a partition, we will say that PV is autonomous with respect to A.

Remark 6.2. If PV is *autonomous with respect to* A *and* B *together* (*i.e.*, "Let $r \in \Pi_{\{ij\}} \subset \mathcal{R}$ iff $A(r) = i$ and $B(r) = j$"), then PV is both autonomous with respect to A and autonomous with respect to B. The converse of this is not true.

Definition 6.3. The *autonomy* of a path-vector system PV is

$$\mathcal{A}(PV) = \{\Pi \mid PV \text{ is autonomous with respect to } \Pi\}$$

One intuitive definition for the concept of full autonomy might be that PV is autonomous with respect to all possible predicates Π. However, this is not reachable. To give a more useful definition, we first introduce the following concept.

Definition 6.4. $Q(r,\ v)$ is an *importability predicate* iff $Q(r,\ v)$ holds if t^{in} applies some $F^{in}(v,u)$ to $r \in X \subset \mathcal{R}$.

Definition 6.5. PV has *full autonomy* iff there exists a PL such that for all instances I over $(PV,\ PL)$ and all vertices v in the instance graph there exists an importability predicate $\mathrm{Imp}(r,v)$ such that for all partitions Π of $\{r \in \mathcal{R} \mid \mathrm{Imp}(r,v)\}$, PV is autonomous with respect to Π.

This definition of full autonomy is more reasonable because it includes node independence and limits the scope of path descriptors considered to those that are actually imported at a given node. Informally then, a path-vector system has full autonomy when imported path descriptors can be ranked freely at every node.

We now define a more specific notion of autonomy suitable for BGP-like systems. It describes the ability to classify neighbors, *e.g.*, so that an ISP can prefer routes from customers over routes from peers.

Definition 6.6. The path-vector policy system $(PV,\ PL)$ supports *autonomy of neighbor ranking* if, for every instance I, node v, and a partition $C_1,\ C_2, \ldots, C_k$ of the set of neighbors of v, there exists a legal import policy at v that does not filter routes such that, for $1 \leq j \leq k-1$, v always prefers routes sent from partition C_j over those sent from partition C_{j+1}.

Note that autonomy of neighbor ranking is simply autonomy with respect to a partition on the value of the next hop (or path vector) attribute of "importable" path descriptors.

The system PV_{up} in Example 5.14 does not support autonomy of neighbor ranking. However the system PV_{force} in Example 5.15 does, but in what might be called a *draconian* manner, *i.e.*, the policy-application functions enforce increasing rank even if the policy writer's policies do not—routes that are not increasing in rank are simply filtered out by the protocol (not the policies).

6.2 Protocol Transparency

This brings us to another important property for policy writers: they should be able to easily understand the semantics of policies that they write. For example, the import-policy application $Y = t^{in}(v,\ u,\ f,\ X)$ is defined with the user-supplied policy f as input, but there is no guarantee that the policy writer can easily understand why the output Y is obtained.

Definition 6.7. Suppose there exists a function $\hat{t}^{in}$ whose definition does not depend on f, such that $t^{in}(v,\ u,\ f,\ X) = f(\hat{t}^{in}(v,\ u,\ X))$. Then PV is said to apply import policies *transparently*. Similarly, if there exists a function $\hat{t}^{out}$ such that $t^{out}(v,\ u,\ f,\ X) = \hat{t}^{out}(v,\ u,\ f(X))$, then PV is said to apply export policies *transparently*. If both of these conditions hold, then PV is *transparent*. In this case, we can define the function $t(v,\ u,\ X) = \hat{t}^{in}(v,\ u,\ \hat{t}^{out}(u,\ v,\ X))$ and note that

$$F_{(v,\ u)}(X) = F^{in}(v,\ u)(t(v,\ u,\ F^{out}(u,\ v)(X))).$$

That is, the transformation between two neighboring nodes participating in PV can be easily understood as the composition of three functions: the export policy at one node; a fixed, uniform transformation t given by PV; and the import policy at another node.

Remark 6.8. The system PV_{force} is not transparent, but the systems PV_{up} and $PV_{\mu bgp}$ are.

6.3 A Design Trade-off

We saw that the systems PV_{up} and PV_{force} are both robust, yet one supports autonomy of neighbor ranking but is not transparent while the other is transparent but does not support autonomy of neighbor ranking. This is just one example of a more general design trade-off:

Theorem 6.9. *If $(PV,\ PL)$ is any path-vector policy system with $\mathcal{M}(PV,\ PL) = \mathcal{APOSPP}$, then either $(PV,\ PL)$ does not support autonomy of neighbor ranking or PV is not transparent, or both.*

Proof. The SPP instance GOOD GADGET in Figure 5(a) is in $\mathcal{APOSPP}$, so it must be expressible by some $(PV,\ PL)$ instance. If $(PV,\ PL)$ supports autonomy of neighbor ranking, then node 2 can change its policies to prefer paths through node 3, producing the SPP instance BAD GADGET in Figure 5(b) which has no solution. Therefore, because $\mathcal{M}(PV, PL) = \mathcal{APOSPP}$, the policy-application functions of PV must not allow this policy to take effect, *i.e.*, the system is not transparent. $\square$

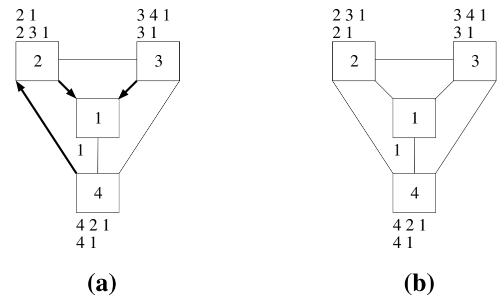

Figure 5: (a) The SPP GOOD GADGET and its unique solution. (b) The SPP BAD GADGET.

6.4 Policy Opaqueness

Policy writers might often think of autonomy and transparency in terms of path-descriptor attributes. In particular, a policy writer might be concerned with what freedom he or she has to change a path-descriptor attribute and what effect such a change might have. A related concern, the property of *policy opaqueness* that we discuss in this section, is whether attribute settings are shared with neighbors or kept private. On one hand, the exchange of information might be important to allow policy writers to make important conditional assignments that affect ranking or the overall robustness of the system; on the other hand, policy writers may not want to disclose their changes to path-descriptor settings (especially when these changes should not influence others).

Informally, an opaque system is one where policy-related attributes are kept hidden when path descriptors are exchanged between nodes. It is expected that this "information hiding" occurs in the protocol transform functions (specifically t^{out}, because we expect t^{in} to be executed by a router that is different than the one that last set attribute values) as a built-in transformation to the path descriptor. So that we may conveniently discuss the opaqueness of a system in terms of which attributes are shared and which are kept private, we make the following definition. Let r^{-A} be the path descriptor r with attribute A removed.

Definition 6.10. Attribute A is *opaque* iff, for any two $r_1, r_2 \in \mathcal{R}$, $r_1^{-A} = r_2^{-A}$ implies that

$$t^{out}\left(v, u, F^{out}(v,u), \{r_1\}\right) = t^{out}\left(v, u, F^{out}(v,u), \{r_2\}\right)$$

for all v, u (*i.e.*, either r_1 and r_2 are both filtered or they produce the same descriptor).

An opaque attribute, then, is one that is essentially cleared on export (after application of t^{out}).

Remark 6.11. The local-preference attribute is opaque in the systems PV_{force} and $PV_{\mu bgp}$, but not in the system PV_{up}. In this case, the opaqueness of local-preference and autonomy of neighbor ranking are closely intertwined because adjusting rank for next-hop involves adjusting the local-preference value accordingly; this is not arbitrarily permitted in PV_{up}. It is the implementation of ranking restrictions in PV_{up} that removes the opaqueness of local preference. It is not generally true that loss of autonomy of neighbor ranking goes hand-in-hand with a loss of opaqueness.

7 Global Constraints

Theorem 6.9 shows that the expressive power of $\mathcal{APOSPP}$ can be reached only if a path-vector policy system gives up either transparency or some autonomy. However, both of these may be very important in many applications. In this section, we discuss an approach that will allow us to move beyond this dilemma: relying on global assumptions in the network.

The expressive power of a path-vector policy system is largely dictated by the *local constraints* included in the specification and those enforced by the policy language. We introduce the complementary notion of a *global constraint* as any function κ that maps any $(PV,\ PL)$ instance I to $\{\text{TRUE},\ \text{FALSE}\}$.

Definition 7.1. A *globally constrained* path-vector policy system is a triple $(PV,\ PL,\ \kappa)$, where κ is a global constraint for $(PV,\ PL)$. I is a *legal instance* of $(PV,\ PL,\ \kappa)$ if I is an instance of $(PV,\ PL)$ and $\kappa(I) = \text{TRUE}$.

Definition 7.2. Let $\mathcal{M}(PV,\ PL,\ \kappa)$ be the set

$$\{\mathcal{E}(S) \mid S \in \mathcal{S}(I) \text{ for a legal } (PV,\ PL) \text{ instance } I\}.$$

Definition 7.3. Define the constraint κ_{apo} as

$$\kappa_{apo}(I) \Leftrightarrow \forall\, S \in \mathcal{S}(I),\ \mathcal{E}(S) \in \mathcal{APOSPP}.$$

We say that the global constraint κ is *robust* for $(PV,\ PL)$ if, for every instance I, $\kappa(I)$ implies $\kappa_{apo}(I)$.

The following theorem implies that global constraints are indeed an integral part of path-vector-system design.

Theorem 7.4. *Suppose the global constraint κ is robust for a transparent $(PV,\ PL)$ allowing autonomy of neighbor ranking such that $\mathcal{M}(PV_{sp}) \subsetneq \mathcal{M}(PV, PL, \kappa)$ (*i.e.*, at least as expressive as shortest paths). Then κ must be non-trivial.*

Proof. If we are not restricted to shortest-paths routing, then autonomy of neighbor ranking and transparency allow us to express BAD GADGET. Only a non-trivial global constraint could prevent this. ☐

8 An Application: Class-Based Path-Vector Policy Systems

The Hierarchical-BGP points in the design space (HBGP, *etc.*), motivated by [5, 6], are examples of a general class of transparent systems where some type of autonomy of neighbor ranking is relevant: route transformations depend on the partition of neighbors into *classes*. We will refer to systems that use a generalized version of such a policy language as *class-based systems*. Theorem 7.4 tells us that such systems require a nontrivial global constraint; in this section we sketch design guidelines for these systems.

8.1 The Class-Based Path-Vector System

We fix a BGP-like path-vector system that can implement *scoping* and *relative preference* rules dictated by class relationships (such as those in [5, 6]). By *scope*, we mean the conditions under which routes are shared with neighbors, and by *relative preference*, we mean the difference in rank assigned to routes learned from neighbors in different classes.

In our running-example system $PV_{\mu bgp}$, path descriptors r contain a *local preference* attribute $l(r)$ that can be set to assign rank based on the class of the exporting neighbor. This attribute is not shared between nodes, intuitively allowing some autonomy and opaqueness. Limited scoping can be implemented by filtering routes. However, this notion of scope is restrictive, *e.g.*, it does not allow easy flagging of a backup route, especially when the next hop might be through a neighbor of an ordinarily preferred class. Therefore, we extend the path descriptor r, following [5], to include a *level* attribute $g(r)$. This attribute is nondecreasing and shared and will have precedence in ranking; thus, it can be used to communicate notions of scope that override relative-preference rules encoded in the local-preference attribute.

Remark 8.1. If all nodes agreed on an encoding within local preference for indicating backup routes or some information were shared between nodes, backup-route scoping would be possible in BGP ($PV_{\mu bgp}$) without additional attributes. However, the additional attribute can separate the awkward encoding and information sharing from attributes meant for local use. The original description of HBGP+BU in [5] discussed these same issues.

The components of the path-vector system PV_{cb} that we use for class-based applications are as follows.

$$
\begin{array}{rcl}
\mathcal{R}_{cb} & = & \mathcal{D}_{cb} \times \mathbb{N} \times \mathbb{N} \times \text{Seq}(\mathbb{N}) \\
\mathcal{U}_{cb} & = & \mathbb{N} \times \mathbb{Z} \text{ (lexically ordered)} \\
dest_{cb}(d, g, l, P) & = & d \\
\omega_{cb}(d, g, l, P) & = & (g, -l) \\
\textsc{o}_{cb}(X) & = & (r \in X) \Rightarrow (\exists d \in \mathcal{D}_{cb}, m \in \mathbb{N} \\
 & & \text{such that } r = (d, 0, 0, m)) \\
\textsc{l}^{in}_{cb}(f) & = & (((d', g', l', P') = f(d, g, l, P)) \\
 & & \Rightarrow (g \leq g' \wedge P = P')) \\
\textsc{l}^{out}_{cb}(f) & = & (((d', g', l', P') = f(d, g, l, P)) \\
 & & \Rightarrow (g \leq g' \wedge P = P')) \\
t^{in}_{cb}(u, v, f, X) & = & \{(d, g, l, P) \in f(X) \\
 & & \mid P \text{ is a simple path}\} \\
t^{out}_{cb}(u, v, f, X) & = & \{(d, g, 0, uP) \mid (d, g, l, P) \in f(X)\}
\end{array}
$$

Note that $\textsc{l}^{in}_{cb}$ and $\textsc{l}^{out}_{cb}$ guarantee that the level attribute is nondecreasing and that t^{out}_{cb} guarantees that local preference is not shared. When ranking, a smaller level attribute is first preferred, then higher local preference. Also, note that PV_{cb} is transparent: let $t(v,\ u,\ X) = \{(d\ , g\ ,\ 0,\ uP) \mid (d,\ g,\ l,\ P) \in X$ where uP is a simple path$\}$ in Definition 6.7.

8.2 Class-Based Policy Languages

The second component of design is a policy language capable of expressing scope and relative-preference rules for class-based systems. We first make formal the notion of class relationships. Let $C = \{C_1, C_2, \ldots, C_c\}$ be a set of *classes*. Every node $v \in V$ will have a *class-assignment function* $C^v : V \to C$ that assigns each neighbor of v a class in C. As an example, consider node v in Figure 6. Here, a node v with neighbors u, w, x has assigned classes C_k, C_i, C_j to these neighbors, respectively.

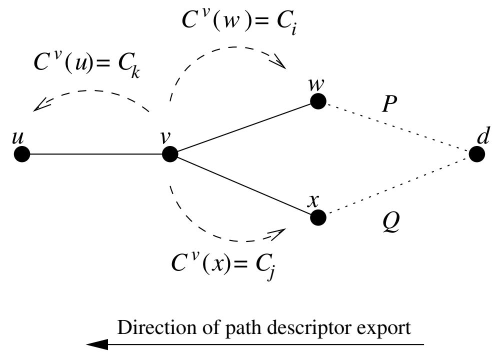

Figure 6: Class assignments to neighbors of node v and paths to a destination node d.

Class assignments might require some consistency, *e.g.*, that "customer" and "provider" assignments occur in consistent pairs; such requirements in a system are expressed by the *cross-class matrix* $X = \{0, 1\}_{c \times c}$. For any pair of nodes $u, v \in V$, if $C^v(u) = C_i$, then $X_{ij} = 1$ if $C^u(v)$ is permitted to be C_j; otherwise, $X_{ij} = 0$.

Remark 8.2. By defintion, X must be symmetric.

Let $(\bullet)$ be the set of *order operators*, *e.g.*, $=, <, \leq$, *etc.*, and $\top$, which means "any relationship," so that $z_1 \top z_2$ is true for any z_1, z_2 in the same ordered set. Relative preference between classes will be described by the *preference matrix* $W = (\bullet)_{c \times c}$ so that if $W_{ij} = \bullet$, then nodes should treat path descriptors r_i, r_j imported from neighbors in classes C_i, C_j, respectively, in a way that ensures $\omega(r_i) \bullet \omega(r_j)$; *e.g.*, in Figure 6, if W_{ij} is $<$, then node v should prefer the path P over the path Q. The policy-language compiler can enforce this as a constraint on local-preference-attribute values set by import policies.

Scope will be described by the *level matrix* $M = ((\bullet) \cup \{\perp\})_{c\times c}$. For any node v and neighbors w, u with $C^v(w) = C_i$ and $C^v(u) = C_k$, if $M_{ik} = \perp$ then for any path descriptor r imported from w, $F^{out}(v,u)(\{r\})$ must equal $\emptyset$. This setting is used to prevent the exchange of routes between classes altogether (filtering); *e.g.*, in Figure 6, if $M_{ik} = \perp$, then v would not export to u any routes it learned from w. Other scoping conditions can be described by allowing or enforcing a change in the level attribute. One example is backup routing: Because lower levels take precedence, a backup route can be assigned a higher level value to avoid being chosen even if it passes through a preferred class. This situation can be sketched using our example Figure 6: Formally, for any node v and two neighbors w, u with $C^v(w) = C_i$ and $C^v(u) = C_k$, assume there is a path P from w to some destination d. Let r_w be the path descriptor at v for the path vP, and let r_u be the path descriptor for the path vP exported to u, *i.e.*, $\{r_w\} = F^{out}(v,u)(\{r_w\})$. The policy complier should enforce through constraints on level-attribute values set in export policies that, if $\bullet = M_{ik}$, then $g(r_w) \bullet g(r_u)$.

Because the level attribute has precedence in ranking over the local-preference attribute, the preference matrix W only applies to descriptors of the same level-attribute value; automatically, lower level values are preferred and this allows descriptors of different levels to be exchanged by neighbors of any class.

Example 8.3. For the system HBGP+BU, let $C = \{C_1,\ C_2,\ C_3\}$, where C_1 can be interpreted "customer," C_2 as "peer," and C_3 as "upstream provider." X should enforce consistent customer-provider and peer-peer relationships; W should enforce that customer routes are preferred over peer routes, and both are preferred over upstream routes; M should enforce that customer routes are shared with all neighbors, and that peer and upstream routes are only shared with customers. In addition, M should permit nodes to flag routes as backup routes so that they are less preferred even if relative preference rules would dictate otherwise. The resulting matrices X, W, and M are as follows.

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$W = \begin{bmatrix} \top & < & < \\ > & \top & < \\ > & > & \top \end{bmatrix}$$

$$M = \begin{bmatrix} \leq & \leq & \leq \\ \leq & < & < \\ \leq & < & \perp \end{bmatrix}$$

A *class description* is the quadruple

$$CD = (C, X, W, M).$$

CD contains all the information necessary to generate a policy language for PV_{cb} whose "compiler," the semantic function $\mathcal{M}$, can generate tuples $(F^{in}, F^{out}, F^{orig})$ from node policies that (1) list class assignments (*i.e.*, C^v) for neighbors and (2) give local preferences and level settings for routes. The tuples will honor the scope and relative preference rules described by CD if the compiler does the following at each node v given its policy configuration p in PL:

1. For all neighbors u, let $F^{in}(v,u)$ set the local preference (and possibly level) attributes of imported path descriptors as specified in the policy configuration p. Check that for all pairs of neighbors u, w, if $C^v(u) = C_i$, $C^v(w) = C_j$, and $\bullet = W_{ij}$, then for all $r_u \in F_{(v,u)}(\mathcal{R}_I^u)$ and $r_w \in F_{(v,w)}(\mathcal{R}_I^w)$, we have that $\omega(r_u) \bullet \omega(r_w)$ if $g(r_u) = g(r_w)$.

2. For all neighbors u, let $F^{out}(v,u)$ set the level of outgoing path descriptors as specified in the policy configuration p. Then check that for all pairs of neighbors u, w, if $C^v(w) = C_i$, $C^v(u) = C_j$, and $\bullet = M_{ij}$, then for all $r \in F_{(v,w)}(\mathcal{R}_I^w)$, $g(r) \bullet g(F^{out}(v,u)(r))$, unless $\bullet = \perp$, in which case $F^{out}(v,u)(r) = \emptyset$.

The policy language can enforce the local constraints described by X, W, and M. Class consistency, along with any further conditions necessary for robustness, must be built into the accompanying global constraint.

Remark 8.4. Class-based systems are autonomous with respect to next-hop class if the descriptors have the same level-attribute value; because a neighbor can be assigned any class, as long as the assignments are consistent, this essentially means that class-based systems have a restricted form of autonomy of neighbor ranking.

8.3 Class-Based Global Constraints

Let the class-consistency constraint $\textsc{c}$ be defined as

$$\forall\, u, v \in V, \\ (C^v(u) = C_i) \Rightarrow (C^u(v) \in \{C_j \in C \mid X_{ij} = 1\}).$$

Let $\textsc{k}_{cb} = \textsc{c} \wedge \textsc{j}$, where $\textsc{j}$ is some constraint such that $\textsc{k}_{cb}$ is robust for PV_{cb} with respect to some PL of the form described above. We now examine how to suitably define the robustness check $\textsc{j}$.

Given the results from Section 5.2, we know that a good starting point for guaranteeing robustness is precluding dispute wheels. Because of the preference and scoping rules associated with class-based systems, we can more easily find potential dispute wheels given the class assignments made by nodes. We first introduce the following helpful result.

Lemma 8.5. *The path descriptors corresponding to all paths R_iQ_{i+1} and Q_i on a dispute-wheel rim in an SPP mapped from a class-based instance have equal level-attribute values.*

Proof. In this proof, SPPs are those mapped from instances of a path-vector system; so, if $P \in \mathcal{P}^v$ in the SPP $S \in \mathcal{S}(I)$, define $d(P) \in \mathcal{R}_{cb}$ as the realizable path descriptor for path P at node v in the path-vector instance I. Recall that $g(r)$ is the level attribute of r.

Assume we have a dispute wheel in some SPP $S \in \mathcal{S}(I)$; then for all i, $\lambda^{v_i}(R_iQ_{i+1}) < \lambda^{v_i}(Q_i)$, so $\omega(d(R_iQ_{i+1})) < \omega(d(Q_i))$; this means that $g(d(R_iQ_{i+1})) \le g(d(Q_i))$. Level is nondecreasing, so $g(d(Q_{i+1})) \le g(d(R_iQ_{i+1}))$. These two inequalities imply that $g(d(Q_{i+1})) \le g(d(Q_i))$ for all i; iterating around the wheel yields $g(d(Q_i)) \le g(d(Q_{i+1}))$, thus $g(d(Q_i)) = g(d(Q_{i+1})) = g(d(R_iQ_{i+1}))$. □

Let $e = \{v,\ u\} \in E$ and let $C^v(u) = C_i$. If e is on a dispute wheel rim, then by Lemma 8.5, there must be a class assignment of another node w by v such that v can export to u a path descriptor from w without increasing the level attribute. But when an edge lies on a dispute wheel rim, it imports a descriptor from two nodes, one along a spoke edge and one also on the rim; so, this condition is true for both the node adjacent to the spoke edge leading to v and for the node adjacent to the rim edge leading to v. This condition, in turn, applies to the rim edge $\{w,\ v\}$ as well (a dispute wheel must contain at least two distinct directed edges), but we cannot iterate further around the wheel because w could import from rim edge e. However, we have just proved that the following statement must hold for any rim edge e:

Lemma 8.6. *If $e = \{v,\ u\} \in E$ with $C^v(u) = C_i$ is on a dispute-wheel rim, then there exists some*

$$C_j \in \{C_x \mid X_{jx} = 1 \textit{ and } M_{xi} \textit{ permits equality}\}$$

such that

$$\exists k : M_{kj} = 1.$$

We can then use Lemma 8.6 to form a constraint that prevents dispute wheels just based on class assignments:

Theorem 8.7. *Given an instance signaling graph G and class assignments, consider the subgraph H containing only edges $\{v,\ u\}$, $C^v(u) = C_i$, with C_i satisfying the condition in Lemma 8.6. If H is acyclic then there is no dispute wheel.*

Proof. Dispute wheen rims must contain edges satisfying the condition in Lemma 8.6. Thus if the signaling subgraph containing only these edges is acyclic, no cycle of these edges, including a dispute wheel in the general signaling graph, is possible. □

Remark 8.8. The sufficient condition in Theorem 8.7 is unnecessarily strong in most cases. However, if $W = \top_{c\times c}$ then this is the only global constraint we know of that can guarantee no dispute wheel. Furthermore, M often permits the construction of a "homogeneous dispute wheel," one where all class assignments in the direction of export are the same around the rim. The constraint in Theorem 8.7 can be weakened to allow such cycles in the testing subgraph and these cycles can then be checked for separately. This observation is especially important for HBGP+BU, where the only potential dispute wheels are homogeneous, and these cycles are prevented by standard Internet economics (see the following example).

Example 8.9. For the system HBGP+BU, $\textsc{j}$ need only check that no customer-provider cycles exist: A simple case-by-case analysis of possible class assignments, given the constraints in matrices C and M, shows that the only dispute wheels possible are cycles in the customer-provider relationship graph. This follows directly from Lemma 8.5. Consider the other possibilities of edges on the dispute wheel:

1. Suppose we have a rim edge $e = \{v, u\}$ where $C^v(u) = C_3$. Then node v must import from a node w without increasing the level attribute; however, only M_{31} permits equality so $C^v(w) = C_1$. Because only $X_{13} = 1$, we have that $C^w(v) = 3$. If w is on a spoke, then because W prefers routes from C_1 neighbors such as w, the adjacent rim node must also be of class C_1. Thus the only situation is one where the adjacent rim edge to v must have the same assignment as this one; this gives the homogeneous customer-provider cycle.

2. Suppose we have a rim edge $e = \{v, u\}$ where $C^v(u) = C_2$. Just as with the case above, only M_{21} permits equality, and by a similar argument, the adjacent rim edge to v must be a customer-provider edge. But this results in case (1) above where the dispute wheel must have these edges all the way around the rim, which contradicts the assumption that $C^v(u) = C_2$. Thus this edge e cannot be on a dispute wheel.

3. Suppose we have a rim edge $e = \{v, u\}$ where $C^v(u) = C_1$. All values M_{1i} permit equality, so v can import the dispute path descriptors from neighbors of any class. Consider the assigment along the rim edge $e' = \{w, v\}$ adjacent to v. By case (2) above, $C^w(v) \neq C_2$. If $C^w(v) = C_3$ then all dispute wheel edges must have this directed assignment, as in case (1) above, so this contradicts the assumed class assignment along edge e. The only other possibility is that $C^w(v) = C_1$, which would give a customer-provider cycle.

Checking for these customer-provider cycles is tractable; even without an explicit check, the basic economics of the current commercial Internet naturally prevent nodes from being customers or providers of themselves.

9 Open Problems

We have defined the Path-Vector Policy System framework: we identified and formalized dimensions of the protocol design space in a way that highlights the role of policy languages.

Several issues that we discussed require additional work. First, either Conjecture 5.3 must be proven or a broader sufficient condition for robustness should be found. Second, the power of class-based systems must be investigated further; in particular, the robustness check presented in Theorem 8.7 is too strong. It is likely that a closer examination of the preference and scoping rules will give a more reasonable set of constraints that do not "over-protect" against dispute wheels and do not preclude too many robust instances. Third, while we justify the inclusion of global constraints in protocol design, we do not discuss how they are enforced. Distributed algorithms, supplementary protocols, or economic incentives could check global consistency. We can also ask what level of expressiveness can be achieved by an autonomous, transparent, and robust system with an imposed global constraint that can be checked by one of the above methods in polynomial time. Finally, additional useful degrees of autonomy should be identified and analyzed (perhaps in the context of specific routing applications).

We have focused on the static semantics of path-vector systems rather than their dynamic behavior. However, in non-deterministic systems, the static and dynamic semantics may become intertwined, *e.g.*, a node might use some temporal condition to break ties between equally ranked routes from different neighbors in a BGP-like system—a system that prefers more recent routes will have very different semantics than one that prefers older routes. Both non-deterministic systems and their dynamic semantics should be investigated. Furthermore, the static semantics of a path-vector system are independent of the algorithm used to find solutions; we are particularly interested in distributed approaches to this problem.

We have focused on the signaling of routes without discussion of how this corresponds to forwarding in the data plane. For example, in BGP, the signaling graph of Internal BGP (IBGP) need not have any relationship to the forwarding graph (IGP forwarding). Several routing anomalies that are related to this independence in BGP have been described elsewhere in [11]. In general, there will be some interaction between the signaling graph, the physical network supporting this signaling, and the paths in the data plane which are controlled by the paths in the signaling plane. We need a general theory that describes this interaction for path-vector protocols.

References

[1] C. Alaettinoglu, T. Bates, E. Gerich, D. Karrenberg, D. Meyer, M. Terpstra, and C. Villamizar. Routing Policy Specification Language (RPSL). RFC 2280, 1998.

[2] O. Bonaventure and B. Quoitin. Common Utilizations of BGP Community Attribute. Manuscript, 2003.

[3] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. RFC 1997, 1996.

[4] Cisco Field Note. Endless BGP Convergence Problem in Cisco IOS Software Releases. `http://www.cisco.com/warp/public/770/fn12942.html`, October 2001.

[5] L. Gao, T. G. Griffin, and J. Rexford. Inherently Safe Backup Routing with BGP. In *Proc. IEEE INFOCOM 2001*, 1:547–556, April 2001.

[6] L. Gao and J. Rexford. Stable Internet Routing without Global Coordination. In *Proc. ACM SIGMETRICS*, pages 307–317, June 2000.

[7] R. Govindan, C. Alaettinoglu, G. Eddy, D. Kessens, S. Kumar, and W. Lee. An Architecture for Stable, Analyzable Internet Routing. *IEEE Network*, 13(1):29–35, 1999.

[8] T. G. Griffin, A. D. Jaggard, and V. Ramachandran. Design Principles of Policy Languages for Path Vector Protocols. In *Proc. ACM SIGCOMM'03*, August 2003.

[9] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Transactions on Networking*, 10(2):232–243, April 2002.

[10] T. G. Griffin and G. Wilfong. An Analysis of BGP Convergence Properties. In *Proc. ACM SIGCOMM'99*, pages 277–288, September 1999.

[11] T. G. Griffin and G. Wilfong. On the Correctness of IBGP Configuration. In *Proc. ACM SIGCOMM'02*, August 2002.

[12] B. Halabi. *Internet Routing Architectures*. Cisco Press, 1997.

[13] C. Hendrick. Routing Information Protocol (RIP). RFC 1058, 1988.

[14] C. Huitema. *Routing in the Internet*. Prentice Hall, 1995.

[15] G. Huston. Interconnection, Peering and Settlements: Part I. *Internet Protocol Journal*, 2(1):2–16, March 1999.

[16] G. Huston. Interconnection, Peering and Settlements: Part II. *Internet Protocol Journal*, 2(2):2–23, June 1999.

[17] G. Huston. Scaling Interdomain Routing—A View Forward. *Internet Protocol Journal*, 4(4):2–16, December 2001.

[18] D. McPherson, V. Gill, D. Walton, and A. Retana. BGP Persistent Route Oscillation Condition. Manuscript, 2002.

[19] B. Rajagopalan, J. Luciani, and D. Awduche. IP Over Optical Networks: A Framework. Manuscript, 2003.

[20] S. Ramachandra and D. Tappan. BGP Extended Communities Attribute. Internet Draft, 2001. Work in progress.

[21] Y. Rekhter and T. Li. A Border Gateway Protocol. RFC 1771 (BGP version 4), 1995.

[22] E. Rosen and Y. Rekhter. BGP/MPLS VPNs. RFC 2547, 1999.

[23] J. Rosenberg, H. Salma, and M. Squire. Telephony Routing Over IP (TRIP). RFC 3219. January 2002.

[24] R. P. Stanley. *Enumerative Combinatorics, Vol. 2*. Cambridge University Press, Cambridge, 1999.

[25] J. W. Stewart. *BGP4, Inter-domain Routing in the Internet*. Addison-Wesley, 1998.

[26] K. Varadhan, R. Govindan, and D. Estrin. Persistent Route Oscillations in Inter-domain Routing. *Computer Networks*, 32:1–16, 2000.

[27] Y. Xu, A. Basu, and Y. Xue. A BGP/GMPSL Solution for Inter-domain Optical Networking. Manuscript, 2002.

A Proofs of SPP and Path-Vector Solution Equivalence

Theorem A.1 (4.5). *If π is a solution for $S_{(I,w,r_w)}$, then*

$$\rho_\pi(v) = \bigcup_{P\in\pi(v)} r(P,\ r_w)$$

is a solution for $I(w,\ r_w)$.

Proof. It is clear that for each v, all path descriptors in $\rho_\pi(v)$ are realizable. We must show that for each v, $\rho_\pi(v) = \min(C(\rho_\pi,\ v))$. If $v = w$, then $\rho_\pi(w) = \{(w)\} = \min(C(\rho_\pi,\ w))$ by definition. Suppose that $v \neq w$. We first note that for any $Y \subseteq \mathcal{P}^v$,

$$\begin{aligned} A &= \bigcup_{P\in\min(\lambda^v,\ Y)} r(P,\ r_w) \\ &= \min\left(\bigcup_{P\in Y} r(P,\ r_w)\right) \\ &= B, \end{aligned}$$

because

$$\begin{array}{ll} & r \in A \\ \text{iff} & \{r\} = r(P,\ r_w) \text{ for some } P \in \min(\lambda^v,\ Y) \\ \text{iff} & \{r\} = r(P,\ r_w) \text{ for some } P \in Y \text{ such that for} \\ & \text{every } P' \in Y,\ \lambda^v_{(I,\ w,\ r_w)}(P) \leq \lambda^v_{(I,\ w,\ r_w)}(P') \\ \text{iff} & \text{for some } P \in Y \text{ such that for every } P' \in Y, \\ & \{r\} = r(P,\ r_w), \{r'\} = r(P,\ r_w), \\ & \text{and } \omega(r) \leq \omega(r') \\ \text{iff} & \text{for all } r' \in (\bigcup_{P\in Y} r(P,\ r_w)),\ \omega(r) \leq \omega(r') \\ \text{iff} & r \in B. \end{array}$$

Let $Y = \{(vQ \in \mathcal{P}^v \mid \{v,\ u\} \in E \text{ and } Q = \pi(u)\}$. Because π is a solution we have $\pi(v) = \min(\lambda^v,\ Y)$ and we have

$$\begin{array}{ll} & \rho_\pi(v) \\ = & \bigcup_{P\in\pi(v)} r(P,\ r_w) \\ = & \bigcup_{P\in\min(\lambda^v,\ Y)} r(P,\ r_w) \\ = & \min(\bigcup_{P\in Y} r(P,\ r_w)) \\ = & \min(\{r \in \mathcal{R} \mid r \in r(P,\ r_w) \text{ for some } P \in Y\}) \\ = & \min(\{r \in \mathcal{R} \mid r \in r(P,\ r_w) \text{ for some} \\ & P \in \{(vQ \in \mathcal{P}^v \mid \{v,\ u\} \in E \text{ and } Q = \pi(u)\}\}) \\ = & \min(\{r \in \mathcal{R} \mid \{v,\ u\} \in E \text{ and} \\ & r \in \bigcup_{Q\in\pi(u)} r(vQ,\ r_w)\}) \\ = & \min(\{r \in \mathcal{R} \mid \{v,\ u\} \in E \text{ and} \\ & r \in \bigcup_{Q\in\pi(u)} F_{(v,\ u)}(r(Q,\ r_w))\}) \\ = & \min(\{r \in \mathcal{R} \mid \{v,\ u\} \in E \text{ and} \\ & r \in F_{(v,\ u)}(\bigcup_{Q\in\pi(u)} r(Q,\ r_w))\}) \\ = & \min(\{r \in \mathcal{R} \mid \{v,\ u\} \in E \text{ and} \\ & r \in F_{(v,\ u)}(\rho_\pi(u))\}) \\ = & \min(C(\rho_\pi,\ v)), \end{array}$$

which completes the proof. ☐

Theorem A.2 (4.6). *If ρ is a solution for $I(w,\ r_w)$, then*

$$\pi_\rho(v) = \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \rho(v)\}$$

is a solution for $S_{(I,w,r_w)}$.

Proof. We need to show that for each v we have $\pi_\rho(v) = \min(\lambda^v,\ \text{candidates}(v, \pi_\rho))$. Because ρ is a solution for $I(w,\ r_w)$, we know that $\rho(v) = C(\rho,\ v) = \min(F^{orig}(v) \cup Y)$, where

$$Y = \{r \in \mathcal{R} \mid \{v,\ u\} \in E \text{ and } r \in F_{(v,\ u)}(\rho(u))\}.$$

It is easy to show that for any X we have

$$\{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \min(X)\} = \min(\lambda^v, \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq X\}).$$

When $v \neq w$, then

$$\begin{aligned}
& \pi_\rho(v) \\
= \ & \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \rho(v)\} \\
= \ & \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \min(Y)\} \\
= \ & \min(\lambda^v, \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \\
& \{r \in \mathcal{R} \mid \{v,\ u\} \in E \text{ and } r \in F_{(v,\ u)}(\rho(u))\}\}) \\
= \ & \min(\lambda^v,\ \{(vQ \in \mathcal{P}^v \mid \{v,\ u\} \in E \text{ and} \\
& Q = \{P' \in \mathcal{P}^v \mid r(P',\ r_w) \subseteq \rho(u)\}\}) \\
= \ & \min(\lambda^v,\ \{(vQ \in \mathcal{P}^v \mid \{v,\ u\} \in E \text{ and} \\
& Q = \pi_\rho(u)\}) \\
= \ & \min(\lambda^v,\ \text{candidates}(v, \pi_\rho))
\end{aligned}$$

When $v = w$, note that $\rho(v) = \{r_w\}$, so we have $\pi_\rho(v) = \{P \in \mathcal{P}^v \mid r(P,\ r_w) \subseteq \{r_w\}\} = \{(w)\} = \min(\lambda^v,\ \text{candidates}(v, \pi_\rho))$. □

Theorem A.3 (4.7). *$\pi_{\rho_\pi} = \pi$ and $\rho_{\pi_\rho} = \rho$.*

Proof.

$$\begin{aligned}
& \pi(v) \\
= \ & \{P \in \mathcal{P}^v \mid \emptyset \neq r(P,\ r_w) \subseteq \textstyle\bigcup_{Q \in \pi(v)} r(Q,\ r_w)\} \\
= \ & \{P \in \mathcal{P}^v \mid \emptyset \neq r(P,\ r_w) \subseteq \rho_\pi(v)\} \\
= \ & \pi_{\rho_\pi}(v)
\end{aligned}$$

$$\begin{aligned}
& \rho(v) \\
= \ & \textstyle\bigcup_{P \in (\{P \in \mathcal{P}^v \mid \emptyset \neq r(P,\ r_w) \subseteq \rho(v)\})} r(P,\ r_w) \\
= \ & \textstyle\bigcup_{P \in \pi_\rho(v)} r(P,\ r_w) \\
= \ & \rho_{\pi_\rho}(v).
\end{aligned}$$

□

B Topologically Sorting SPPs

Theorem B.1. *If $S \in \mathcal{APOSPP}$ then there exists an instance $S' \in \mathcal{ISPP}$ such that $S' \in \mathcal{E}(S)$.*

Proof. We give an iterative process converging to a path-ranking function Λ that is increasing.

Define the *path-rank function for node v at step k* to be λ_k^v. For all $v \in V$ and $P \in \mathcal{P}^v$, let $\lambda_k^v(P) = \infty$ for all $k \leq 0$. For $k > 0$, define λ_k^v as follows: At every node $v \neq v_0$, consider exactly the paths permitted at v, $\mathcal{P}^v$, which have the form vuP', where either $u = v_0$ and $P' = \epsilon$ or $u \neq v_0$ and $uP' \in \mathcal{P}^u$. List these in decreasing order of preference as $P_1 = vu_1P_1',\ P_2 = vu_2P_2',\ \ldots,\ P_i = vu_iP_i'$. (Ties can be broken arbitrarily.) If $u_1 = v_0$, then let

$$\lambda_{k+1}^v(P_1) = 1,$$

and if $u_1 \neq v_0$ let

$$\lambda_{k+1}^v(P_1) = \lambda_k^{u_1}(P_1') + 1.$$

For the less preferred paths P_j, $2 \leq j \leq i$, if $u_j = v_0$, let

$$\lambda_{k+1}^v(P_j) = \lambda_k^v(P_{j-1}) + 1,$$

and for $u_j \neq v_0$ let

$$\lambda_{k+1}^v(P_j) = \max\left\{\lambda_k^{u_j}(P_j'),\ \lambda_{k+1}^v(P_{j-1})\right\} + 1.$$

Assume that all undefined values of λ are ∞ in the above.

Assuming that the set of permitted paths is closed under the taking of subpaths, if the longest permitted path in the SPP has k edges, then for all $v \in V$ and for all $P \in \mathcal{P}^v$, $\lambda_{k'}^v(P) \neq \infty$ for every $k' \geq k$. The path-rank functions will stabilize over iterations if the SPP S is almost-partially ordered, so in S', let

$$\Lambda(v) = \lim_{k \to \infty} \lambda_k^v.$$

Note that in using the above iterative process, ranks are always set higher than neighboring ranks because of the increment used in defining λ_k^v. Indeed, $\lambda^v(vuP) > \lambda^u(uP)$ after convergence, thus Λ and S' are increasing.

Finally, it is clear that $S' \in \mathcal{E}(S)$, because the ranking given by the converging import functions is consistent with the SPP preference list at every node. □

Remark B.2. Any almost-partially ordered SPP can be convered to an increasing SPP using the method described above. It can also be shown that an SPP which cannot converge with respect to the above process (*i.e.*, for some $P \in \mathcal{P}^v$, there does not exist any integer k' such that $\lambda_k^v(P) \neq \infty$ for $k \geq k'$) must have a dispute wheel and thus is not almost-partially ordered.