Assigning AS Relationships to Satisfy the Gao-Rexford Conditions

Luca Cittadini*, Giuseppe Di Battista*, Thomas Erlebach†, Maurizio Patrignani*, and Massimo Rimondini*
* Dept. of Computer Science and Automation, Roma Tre University
{ratm,gdb,patrigna,rimondin}@dia.uniroma3.it
† Dept. of Computer Science, University of Leicester
t.erlebach@mcs.le.ac.uk

***Abstract*—Compliance with the Gao-Rexford conditions [1] is perhaps the most realistic explanation of Internet routing stability, although BGP is renowned to be prone to oscillations. Informally, the Gao-Rexford conditions assume that (*i*) the business relationships between Internet Service Providers (ISPs) yield a hierarchy, (*ii*) each ISP behaves in a rational way, i.e., it does not offer transit to other ISPs for free, and (*iii*) each ISP ranks routes through customers better than routes through providers and peers.**

We show an efficient algorithm that, given a BGP configuration, checks whether there exists an assignment of peer-peer and customer-provider relationships that complies with the Gao-Rexford conditions. Also, we show that preferring routes through peers to those through providers, although more suitable than the original formulation of Condition (*iii*) to describe the business relationships between ISPs, makes the problem NP-hard.

The above results hold both in the clean theoretical framework introduced in [2] to model BGP and in a more realistic setting where (*i*) *local preferences* are assigned on a per-neighbor basis and (*ii*) transit is allowed from/to specific neighbor pairs. Observe that the latter setting, where policy complexity only depends on the number of neighbors, is very close to the way in which operators typically configure routers.

I. Introduction and Related Work

Internet interdomain routing relies on the Border Gateway Protocol (BGP) [3]. Unfortunately, it has been shown that the interaction of not-so-unlikely BGP configurations can lead to permanent oscillations of Internet routing (see, e.g.,[2]). Even worse, deciding whether a given BGP configuration is *safe*, that is, free from oscillations, poses computationally difficult problems (see, e.g., the game theoretical approach in [4] and the stability problems analyzed in [5]).

Hence, safety in interdomain routing can be guaranteed either by extending the protocol to dynamically resolve oscillations [6], [7], or by limiting the expressiveness of the policies that ISPs can configure [8], [2]. Given the popularity of BGP, the former option faces serious deployment issues. In the context of the latter option, it is key that constraints on policies be easy to enforce, or at least easy to check. Jaggard and Ramachandran studied in [9] the special case where policies are class-based, i.e., they are specified based on classes of neighbors. The authors present a polynomial time algorithm that checks whether such class-based policies may potentially lead to oscillations in some specific topologies.

Gao and Rexford proposed in [1] a realistic set of conditions that guarantees safety. A fundamental property of the Gao-Rexford conditions is that they allow router configurations that are consistent with simple economic requirements. In fact, the conditions assume that the business relationships between ISPs yield a hierarchy (*acyclicity* condition), and that each ISP behaves in a rational way, i.e., it does not offer transit to other ISPs for free (*valley-free* condition), and it ranks routes according to revenues (*prefer-customer* condition). Those business arguments led the Internet community to regard compliance with the Gao-Rexford conditions as the intrinsic reason why BGP is actually able to make the Internet work.

A BGP Internet-like network which is compliant with the Gao-Rexford conditions also has other properties beyond safety: for example, it is safe even under router or link failures [10], and it has a bound on convergence time that, assuming that customer-provider chains are of limited length, is roughy constant [11]. In [10] an extension to the Gao-Rexford conditions is proposed that guarantees safety even when interdomain backup policies are in place.

Research interest in the complexity of checking the Gao-Rexford conditions was originally spurred by the findings in [12]. In [13] is devised a polynomial-time algorithm that, given a set of paths, assigns business relationships in such a way that all the paths satisfy the *valley-free* condition. In [14] is shown that assigning relationships while satisfying the *acyclicity* and *valley-free* conditions together takes polynomial time in the general case. Both [13] and [14] regard compliance with the Gao-Rexford conditions as a strong indication that the inferred business relationships are realistic.

In this paper we aim at using the Gao-Rexford conditions to prove that a given BGP network exhibits the highly desirable stability properties listed above. More formally, given a set of BGP policies, we study the problem of deciding whether an assignment of relationships exists that satisfies all the Gao-Rexford conditions. Notice that this is much different from (and harder than) detecting violations to the Gao-Rexford conditions when the relationships are known (see, e.g., [15]). On the other hand, the assignment we compute (if any exists) could differ from the actual business relationships in the Internet: indeed, business relationships might be much more complicated than the simple peer-peer and customer-provider envisaged in the Gao-Rexford conditions. However, we stress that our focus is on verifying the stability properties of the network by means of the Gao-Rexford conditions, and *not* on inferring realistic business relationships.

Observe that collecting Internet-wide BGP router configurations is of course unfeasible, posing both technical and political problems. However, verifying that a given set of policies complies with the Gao-Rexford conditions is still a relevant problem in the context of network simulators (e.g., [16]). Also, there are situations (see, e.g., BGP confederations [17]) where several BGP neighbors work together to define BGP policies that fit a diverse set of interests. In that case the routing policies of the confederation members are likely to be shared.

We show an efficient algorithm that is able to check whether, for a given set of BGP policies, there exists an assignment of relationships that complies with the Gao-Rexford conditions. We use the theoretical framework introduced in [2] to model BGP policies, because it is expressive enough to capture virtually any routing policy, including, e.g., those resulting from the use of BGP communities. We also prove our results in a more realistic setting where (*i*) *local preferences* are assigned on a per-neighbor basis and (*ii*) transit is allowed from/to specific neighbors (e.g., traffic from ISP A can go through ISP B and cannot go through ISP C). Observe that the latter setting, where policy complexity only depends on the number of neighbors, is very close to the way in which operators typically configure routers. An interesting consequence of our results is that, if reliable and accurate shared databases of routing policies (which was the original goal of Internet Routing Registries [18]) were available, then we would be able to efficiently test the stability of the whole system.

The *prefer-customer* condition of [1] specifies that a routing policy should prefer routes through customers over those through peers and providers. However, ISPs are also likely to prefer routes through peers over those through providers, because of lower costs. Such a recasting of the conditions, while preserving routing safety, better captures the economic rationality of ISPs and one could wonder why the original conditions have been stated as *prefer-customer* only. We show that this question has a computational complexity answer. Namely, we show that, given a BGP configuration, it is NP-hard to check if there exists an assignment of relationships that complies with the Gao-Rexford conditions augmented with a *prefer-peer* guideline. Even this result is shown to hold both in the theoretical framework in [2] and in a more realistic setting.

The rest of the paper is organized as follows. In Section II we introduce the models we use to represent BGP policies and we recall the Gao-Rexford conditions. Section III describes an efficient algorithm to check whether a BGP configuration complies with the Gao-Rexford conditions. In Section IV we show that preferring peers to providers makes the same check computationally hard. Conclusions and open problems are discussed in Section V.

II. A Model for BGP Configurations

In this paper we model BGP using the widely adopted *Stable Paths Problem* (SPP) formalism [2], that is the reference point of most of the scientific contributions on BGP stability.

A. The SPP Model

Let $G = (V, E)$ be an undirected graph, with vertex set $V = \{0, 1, \ldots, n\}$ and edge set E. Graph G is used to represent the Internet topology at the level of the Autonomous Systems (ASes). Vertices in V correspond to ASes, while edges in E correspond to adjacency relationships between ASes (also called *peerings*).

A *path* P in G is a sequence of $k+1$ vertices $P = (v_k\ v_{k-1}\ \ldots\ v_1\ v_0)$, $v_i \in V$, such that $(v_i, v_{i-1}) \in E$ for $i = 1, \ldots, k$. Vertex v_{k-1} is the *next hop* of v_k in P. For $k = 0$ we obtain the trivial path (v_0) consisting of vertex v_0 alone. We denote the empty path by ϵ. The *concatenation* of two nonempty paths $P = (v_k\ v_{k-1}\ \ldots\ v_i)$, $k \geq i$, and $Q = (v_i\ v_{i-1}\ \ldots\ v_0)$, $i \geq 0$, denoted as PQ, is the path $(v_k\ v_{k-1}\ \ldots\ v_i\ v_{i-1}\ \ldots\ v_0)$. We assume that $P\epsilon = \epsilon P = \epsilon$, that is, the empty path can never extend or be extended by other paths.

In the SPP model each vertex in $V - \{0\}$ attempts to establish a path to a single vertex 0. Since BGP manages each destination independently from the others, we assume that vertex 0 is the only destination in the network.

As pointed out by several authors (see, e.g. [8]), BGP policies consist of *filtering* and *ranking* components. To model BGP route filters, each vertex $u \in V$ is assigned a set of *permitted paths* $\mathcal{P}^u$. All the paths in $\mathcal{P}^u$ are simple (i.e., without repeated vertices), start from u and end in 0, and represent the paths that u can use to reach 0. The empty path represents unreachability of 0 and is permitted at each vertex $u \neq 0$. Let $\mathcal{P}^0 = \{(0)\}$, that is, vertex 0 can reach itself only directly. Let $\mathcal{P} = \bigcup_{u \in V} \mathcal{P}^u$. To model BGP path ranking, for each vertex $u \in V$, a *ranking function* $\lambda^u : \mathcal{P}^u \rightarrow \mathbb{N}$ determines the relative level of preference $\lambda^u(P)$ assigned by u to path P. If $P_1, P_2 \in \mathcal{P}^u$ and $\lambda^u(P_2) < \lambda^u(P_1)$ (or, more informally, $P_2 < P_1$), then P_2 is *preferred* over P_1. Let $\Lambda = \{\lambda^u | u \in V\}$.

According to the model in [2], the following conditions hold on the paths, for each vertex $u \in V - \{0\}$:

(*i*) $\forall P \in \mathcal{P}^u, P \neq \epsilon$: $\lambda^u(P) < \lambda^u(\epsilon)$ (unreachability of 0 is the last resort);

(*ii*) $\forall P_1, P_2 \in \mathcal{P}^u, P_1 \neq P_2 : \lambda^u(P_1) = \lambda^u(P_2) \Rightarrow P_1 = (u\ v)P'_1, P_2 = (u\ v)P'_2$, (strict ranking is assumed on all the paths but those with the same next hop).

An instance S of SPP is a triple $(G, \mathcal{P}, \Lambda)$. An example is shown in Fig. 1, using the same graphical convention as in [2]. The list beside each vertex u represents the paths in $\mathcal{P}^u$ sorted by increasing values of λ^u. The empty path and $\mathcal{P}^0$ are omitted for brevity. In the example, vertex 2 can use either path in $\mathcal{P}^2 = \{(2\ 1\ 0), (2\ 0)\}$ to reach 0 and prefers $(2\ 1\ 0)$.

A *path assignment* is a function π that maps each vertex $v \in V$ to a path $\pi(v) \in \mathcal{P}^v$, thus modeling the paths selected by BGP. We have that $\pi(0) = (0)$ and, if $\pi(v) = \epsilon$, then v cannot reach vertex 0.

In order to model the BGP protocol dynamics, π can be updated according to a distributed algorithm known as *Simple Path Vector Protocol* (SPVP) [6]. Very briefly, SPVP works

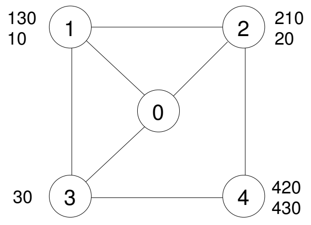


Fig. 1: Example of an SPP instance.

as follows (the details can be found in [6]). Vertex 0 keeps announcing to its neighbors about its presence. Every other vertex u first of all collects announcements from its neighbors and discards those announcements containing paths that are not in $\mathcal{P}^u$. Thus, u can select a path in the following set:

$$\text{choices}(\pi, u) = \begin{cases} \{(u\ v)\pi(v) \mid (u,v) \in E\} \cap \mathcal{P}^u & \text{if } u \neq 0 \\ \{(0)\} & \text{if } u = 0 \end{cases}$$

Let W be the set of paths received by u from its neighbors. At this point, u selects the best ranked path in W according to its ranking function λ^u:

$$\text{best}(W, u) = \begin{cases} \underset{P \in W}{\arg\min}\ \lambda^u(P) & \text{if } W \neq \oslash \\ \epsilon & \text{if } W = \oslash \end{cases}$$

If this operation updated u's selected path, then u sends announcements to all its neighbors advertising path $(u)P$.

Given an SPP instance, we say that π is a *stable path assignment* if, $\forall u \in V$: $\pi(u) = \text{best}(\text{choices}(\pi, u), u)$, that is, every vertex has settled to the best possible choice and has no better ranked alternative. It has been shown that, possibly depending on the timings with which announcements are exchanged, the SPVP algorithm might oscillate indefinitely, never converging to a stable state. An SPP instance S is *safe* [19], [1] if SPVP is guaranteed to converge on S, regardless of the event timings.

Since vertices and edges that are not used by any paths in $\mathcal{P}$ can never be part of any BGP-computed routing path, we assume that the *size* of an SPP instance S is the size of $\mathcal{P}$.

B. The Gao-Rexford Conditions

ISPs establish commercial agreements [1] with each other in order to get the level of connectivity they require, and graph G can be partially oriented to represent these agreements. Namely, edge (u, v) is oriented from u to v (written $(u \to v)$), if u is a *customer* of v, while (u, v) is unoriented (written $(u — v)$) if u and v are *peers*. A graph with these features is a *customer-provider graph*. The neighbors of each vertex v are partitioned into sets $customers(v)$, $providers(v)$, and $peers(v)$, containing neighbors w such that w is a customer, a provider, or a peer of v, respectively.

Assuming that each AS configures its routers in an economically rational way, commercial agreements established among ASes imply some constraints on BGP policies. In particular, we say that *an SPP instance satisfies the Gao-Rexford conditions* [1], [10] if:

(i) *Acyclicity*: The customer-provider graph does not contain a directed cycle (cycle composed by directed edges only). Cycles would correspond to unclear customer-provider roles.

(ii) *Valley-free*: Each path of $\mathcal{P}$ is *valley-free*: provider-to-customer and peer-peer edges can only be followed by provider-to-customer edges. A valley is considered an anomaly because it corresponds to an AS providing transit to either its peers or its providers, hence bearing a cost without getting revenues.

(iii) *Prefer-customer*: For each vertex v, let $P_1, P_2 \in \mathcal{P}^v$. If the next hop of P_1 is in $customers(v)$ and the next hop of P_2 is in $providers(v)$ or in $peers(v)$, then $P_1 < P_2$. This corresponds to preferring routes with lower cost.

In [1], [10] it has been shown that an instance of SPP satisfying all the above conditions is safe. In this sense, Gao and Rexford proposed the above conditions as a methodological guideline to configure routers. In this paper we look at the conditions from a different perspective. Namely, given the BGP configurations of the routers, and assuming no knowledge of the customer-provider relationships, we want to check whether these configurations are compliant with the conditions. More formally, we tackle the following problem:

Problem:	GAO-REXFORD-CHECK
Instance:	An instance $S = (G, \mathcal{P}, \Lambda)$ of SPP.
Question:	Can G be partially oriented to a customer-provider graph such that S satisfies the Gao-Rexford conditions?

The partial orientation of G, if it exists, is a *solution* of GAO-REXFORD-CHECK.

C. A Succinct Model of BGP Routing Policies

The SPP model has the advantage of representing policies in a clear and intuitive way while being expressive enough to model virtually any routing policy (including, e.g., those resulting from the use of BGP communities). On the other hand, such expressiveness requires an explicit representation of all the paths permitted by BGP filters. This implies that the size of S can be exponential in $|V|$. However, in order to keep the complexity of BGP policies manageable, router configuration languages typically use constructs that allow network operators to express policies in a *succinct* way, without the need to enumerate a large number of routes. To show the practical applicability of our techniques, we define a very simple variant of the SPP model that overcomes this problem by representing BGP policies succinctly. We call this variant SSPP.

Let $G = (V, E)$ be defined as for standard SPP instances. To model BGP route filters, for each vertex $u \in V$, we define a set of *permitted path fragments* $\tilde{\mathcal{P}}^u$ such that (i) all path fragments have length at most 3; (ii) path fragments in the form $(u\ 0)$ can be in $\tilde{\mathcal{P}}^u$ only if $(u, 0) \in E$; and (iii) path fragments in the form $(u\ v\ w)$ can be in $\tilde{\mathcal{P}}^u$ only if u, v, and w are distinct vertices in V and $(u, v), (v, w) \in E$.

The only permitted path fragment at vertex 0 is $\tilde{\mathcal{P}}^0 = \{(0)\}$. A vertex $u \in V - \{0\}$ can use any path starting with a fragment in $\tilde{\mathcal{P}}^u$ to reach 0. We implicitly assume that $\epsilon \in \tilde{\mathcal{P}}^u$, $\forall u \in V - \{0\}$. Let $\tilde{\mathcal{P}} = \bigcup_{u \in V} \tilde{\mathcal{P}}^u$.

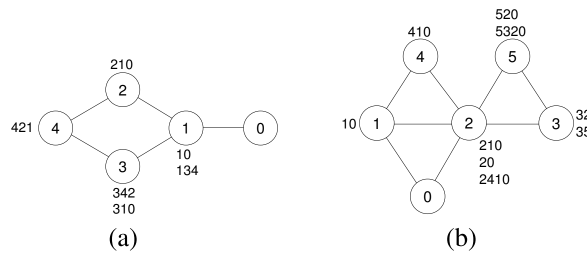


Fig. 2: (a) An instance of the SSPP model for BGP policies. (b) Instance of Gao-Rexford-Check.

Each vertex $u \in V - \{0\}$ ranks path fragments in $\tilde{\mathcal{P}}^u$ according to a function $\tilde{\lambda}^u : \tilde{\mathcal{P}}^u \rightarrow \mathbb{N}$ which determines the level of preference assigned to paths starting with a fragment in $\tilde{\mathcal{P}}^u$. Namely, if $\tilde{\lambda}^u((u\ v\ w)) < \tilde{\lambda}^u((u\ x\ y))$ (or, more informally, $(u\ v\ w)) < (u\ x\ y)$), then any path starting with $(u\ v\ w)$ is preferred over any path starting with $(u\ x\ y)$.

Similarly to the SPP model, unreachability is the last resort, i.e., $\forall P \in \tilde{\mathcal{P}}^u, P \neq \epsilon$: $\tilde{\lambda}^u(P) < \tilde{\lambda}^u(\epsilon)$.

Differently from the SPP model, two path fragments can have the same rank even if they have a different next hop. Moreover, paths through the same neighbor always have the same rank, i.e., let $(u\ v\ w)$ and $(u\ v\ z)$ be two path fragments in $\tilde{\mathcal{P}}^u$, we have $\tilde{\lambda}^u((u\ v\ w)) = \tilde{\lambda}^u((u\ v\ z))$. Finally, any deterministic criterion (e.g., shortest path) can break ties.

An instance $\tilde{S}$ of SSPP is then a triple $(G, \tilde{\mathcal{P}}, \tilde{\Lambda})$.

We remark that SSPP is particularly suitable to capture the semantics of a network where routers have policies that depend only on the neighbors. Such policies envisage only filters of the type "pass all the routes learned from neighbor w to neighbor v", and only rankings of the type "rank routes learned from neighbor x better than those learned from neighbor y".

Whereas sharing significant similarities, SSPP cannot be regarded as a variant of the model presented in [9]. Most noticeably, SSPP is more expressive in that each AS preserves its own autonomy rather than being compelled to rank and filter routes according to predetermined network-wide constraints.

Observe that the size of an SSPP instance is polynomial in $|V|$. In fact, the number of path fragments at each vertex u is bounded by the square of the maximum vertex degree of G.

Consider, for example, the graph in Fig. 2(a). It represents a network where vertices apply the following policies: 1 announces paths received from 0 to 2 and 3; 2 announces paths received from 1 to 4; 3 announces paths received from 4 to 1; 4 announces paths received from 2 to 3; and all the other paths are filtered out. Also, 1 prefers to use the direct path to 0, and 3 prefers paths through its neighbor 4. An instance of SSPP modeling these polices can be built as in Fig. 2(a). It is interesting to notice that the presence of $(3\ 4\ 2)$ in $\tilde{\mathcal{P}}^3$ does not represent an explicit statement of 3, but rather the combined effect of the ranking policy of 3 and of the filtering policy of 4.

The Gao-Rexford conditions can be easily restated in the SSPP model: the *valley-free* and *prefer-customer* conditions are applied to path fragments instead of paths, while the acyclicity condition remains unaltered.

Given a specific tie break criterion, an instance $\tilde{S} = (G, \tilde{\mathcal{P}}, \tilde{\Lambda})$ of SSPP can be uniquely mapped to an instance $S = (G, \mathcal{P}, \Lambda)$ of SPP in the following way. Graph G is unchanged. Given $k+2$ path fragments $(u_1\ v_1\ w_1), (u_2\ v_2\ w_2), \ldots, (u_k\ v_k\ w_k), (v_k\ w_k\ 0), (w_k\ 0)$, such that $u_{i+1} = v_i$ and $v_{i+1} = w_i$ for $i = 1, \ldots, k-1$, we define their *chain* as the path $(u_1\ v_1\ w_1\ w_2\ w_3\ \ldots\ w_k\ 0)$. The set $\mathcal{P}$ consists of all the possible chains of path fragments in $\tilde{\mathcal{P}}$. Sets $\mathcal{P}^u$ are naturally defined as consisting of the paths in $\mathcal{P}$ starting at u. For each vertex $u \in V$, if $\tilde{\lambda}^u((u\ v\ w)) < \tilde{\lambda}^u((u\ x\ y))$, then we require that $\lambda^u((u\ v\ w)P) < \lambda^u((u\ x\ y)Q)$ for any $(u\ v\ w)P, (u\ x\ y)Q \in \mathcal{P}^u$. Ties are then broken according to the deterministic criterion used in the definition of the SSPP instance.

SSPP is clearly less expressive than SPP, yet it admits instances where SPVP fails to converge (see, e.g., the BAD-GADGET example in [2]). We say that an SSPP instance $\tilde{S}$ is *safe* if the SPP instance S obtained by mapping $\tilde{S}$ to S is safe. We now exploit the following property to assess the relationship between the restated Gao-Rexford conditions and the safety of an SSPP instance.

Property 2.1: A path P is valley-free if and only if all the subpaths of P are valley-free.

Lemma 2.1: If an SSPP instance $\tilde{S}$ satisfies the Gao-Rexford conditions, then $\tilde{S}$ is safe.

Proof: We prove the statement by showing that the SPP instance S obtained by mapping $\tilde{S}$ to S satisfies the Gao-Rexford conditions. This is obvious for acyclicity. Since paths in $\mathcal{P}$ are constructed by chaining the valley-free path fragments in $\tilde{\mathcal{P}}$, by Property 2.1 instance S also satisfies the *valley-free* condition. Last, ranking functions in $\tilde{\Lambda}$ enforce the *prefer-customer* condition by applying an appropriate per-neighbor ranking. By construction, the same per-neighbor ranking is retained in the functions in Λ, therefore instance S also satisfies the *prefer-customer* condition. ■

It is therefore interesting to also consider the following problem:

Problem:	Gao-Rexford-Check-Succinct
Instance:	An instance $\tilde{S} = (G, \tilde{\mathcal{P}}, \tilde{\Lambda})$ of SSPP.
Question:	Can G be partially oriented to a customer-provider graph such that $\tilde{S}$ satisfies the Gao-Rexford conditions?

III. A Polynomial Time Algorithm to Check the Gao-Rexford Conditions

In this section we present a polynomial time algorithm for problem Gao-Rexford-Check. Let $S = (G, \mathcal{P}, \Lambda)$ be an instance of SPP, where $G = (V, E)$ is an undirected graph. For a vertex $u \in V$ and two different edges $(u, v), (u, w) \in E$, we let $(u, v) \prec (u, w)$ if vertex u prefers some path starting with $(u\ v)$ to some path starting with $(u\ w)$. More formally, $(u, v) \prec (u, w)$ if there exist paths $P, Q \in \mathcal{P}^u$, $P' \in \mathcal{P}^v$, $Q' \in \mathcal{P}^w$ such that $P = (u\ v)P'$, $Q = (u\ w)Q'$, and $\lambda^u(P) < \lambda^u(Q)$. We assume that $(u, v) \prec (x, y)$ can only hold if $u = x$. Derived from the *prefer-customer* condition, we interpret $(u, v) \prec (u, w)$ as the following constraint: $(u \leftarrow w)$ implies $(u \leftarrow v)$. We refer to all these constraints as the $\prec$ constraints. Furthermore, we consider the transitive closure of $\prec$, i.e., we assume that $(u, v) \prec (u, w)$ and $(u, w) \prec (u, x)$ implies $(u, v) \prec (u, x)$. We also use the expression u *prefers* v *to* x to express $(u, v) \prec (u, x)$.

Let $(G, \mathcal{P}, \prec)$ be a triple consisting of a graph G, a set of paths $\mathcal{P}$ in G, and a transitive binary relation $\prec$ on the edges of G. We assume that some of the edges of G may already have been oriented. We now consider a relaxed variant of the Gao-Rexford conditions in which *prefer-customer* is replaced by the $\prec$ constraints. Namely, we address the following problem:

Problem:	GAO-REXFORD-SIMPLE-CHECK
Instance:	A triple $(G, \mathcal{P}, \prec)$ and an orientation for some edges of G.
Question:	Can G be partially oriented to an acyclic customer-provider graph such that the input orientation is maintained, paths in $\mathcal{P}$ are valley-free, and the $\prec$ constraints are satisfied?

A partial orientation of G that satisfies the above constraints is a *solution* for GAO-REXFORD-SIMPLE-CHECK.

Lemma 3.1: A polynomial-time algorithm for problem GAO-REXFORD-SIMPLE-CHECK gives a polynomial-time algorithm for problem GAO-REXFORD-CHECK.

Proof: Given an instance $S = (G, \mathcal{P}, \Lambda)$ of GAO-REXFORD-CHECK, construct instance $S' = (G, \mathcal{P}, \prec)$ of GAO-REXFORD-SIMPLE-CHECK by keeping G and $\mathcal{P}$ and by defining $\prec$ as above. It is easy to see that a partial orientation of G is a solution for S' if and only if it is a solution for the original instance S. ■

We now describe a polynomial-time algorithm for GAO-REXFORD-SIMPLE-CHECK. Let $S = (G = (V, E), \mathcal{P}, \prec)$ be an instance of GAO-REXFORD-SIMPLE-CHECK.

The idea of the algorithm is to follow the basic approach of Kosub et al. [14]. For a given set of paths $\mathcal{P}$ in an undirected graph G, their algorithm finds an acyclic orientation of G that makes all paths in $\mathcal{P}$ valley-free, if one exists. Their algorithm simply identifies a vertex v that is not traversed by any path (i.e., there is no path in $\mathcal{P}$ such that v is an internal node of the path), orients the edges of v away from v, and recurses on $V - v$. In our algorithm, we process the vertices in the same order, but it is not possible to determine the orientation of the edges as easily. In particular, the $\prec$ constraints force us to deal with the edges incident with v in a much more intricate way. For some of the edges we need to leave the orientation undecided initially, and only after $V - v$ has been oriented, we can also orient these edges. Furthermore, we need to add auxiliary paths to $\mathcal{P}$ before the recursive call in order to ensure that the edges incident with v can be oriented consistently in the end.

The details of our algorithm are as follows. First, the algorithm finds a vertex $v \in V$ that is not traversed by any path (i.e., v does not appear as an internal node of any path in $\mathcal{P}$) and that does not have any edge oriented towards it. If no such vertex exists, the algorithm stops and answers that S is a no-instance. Note that if S is a yes-instance, i.e., there is an acyclic orientation that makes all paths in $\mathcal{P}$ valley-free, there must be at least one vertex that has no incoming edges in that orientation, and that vertex cannot be traversed by any path.

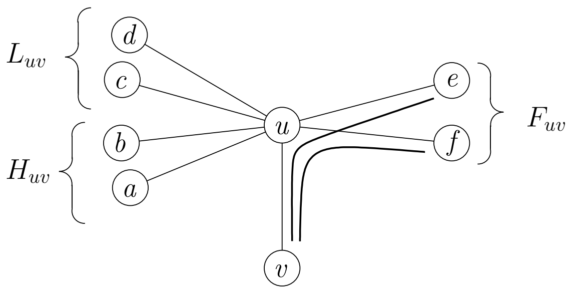


Fig. 3: Illustration of edge sets H_{uv}, L_{uv} and F_{uv} assuming $ua \prec uv$, $ub \prec uv$, $uv \prec uc$, and $uv \prec ud$.

The algorithm aims to orient each edge incident with v away from v (customer-provider edge) or leave it unoriented (peer-to-peer edge). Any such orientation satisfies the $\prec$ constraints at v and cannot create a valley at v. Furthermore, any solution for S in which some edges are oriented towards v is still a valid solution if those edges are made unoriented.

The algorithm considers each edge (v, u) incident with v in turn. The edge to neighbor u is processed as follows. Define the following (not necessarily disjoint) sets of edges incident with u:

- Let H_{uv} be the set of edges $(u, w) \in E$ with $w \neq v$ and $(u, w) \prec (u, v)$. In other words, H_{uv} is the set of edges (u, w) such that u prefers w to v. ('H' indicates *higher* preference.)
- Let L_{uv} be the set of edges $(u, w) \in E$ with $w \neq v$ and $(u, v) \prec (u, w)$. In other words, L_{uv} is the set of edges (u, w) such that u prefers v to w. ('L' indicates *lower* preference.)
- Let F_{uv} be the set of edges $(u, w) \in E$ with $w \neq v$ such that $\mathcal{P}$ contains a path containing edges (v, u) and (u, w). F_{uv} contains edges that follow (v, u) (or are followed by (u, v)) in some path in $\mathcal{P}$.

See Fig. 3 for an illustration. The motivation for defining these edge sets is as follows. The edges in sets H_{uv} and L_{uv} interact with (v, u) because of the $\prec$ constraints, and the edges in F_{uv} because of the valley-free constraints.

In the following, whenever the algorithm decides to assign a certain orientation to an edge (or to make the edge a peer-to-peer edge) and the edge has previously been oriented differently (either in the initial partial orientation or in a previous step of the algorithm), the algorithm outputs 'no' at this stage. In order to simplify the presentation, we do not mention this explicitly in every step.

If the edge (v, u) has already been oriented as $(v \rightarrow u)$, the algorithm orients all edges in H_{uv} towards u (forced by $\prec$), and the processing of the edge (v, u) is finished.

Assume now that (v, u) has not yet been oriented. First, the algorithm checks if $H_{uv} \cap F_{uv} \neq \emptyset$. If so, the algorithm orients every edge (u,w) in $H_{uv} \cap F_{uv}$ as $(u \leftarrow w)$. Any other orientation would create a contradiction, as it implies $(v \rightarrow u)$ (by valley-freeness) and then $(u \leftarrow w)$ (since $(u,w) \prec (u,v)$).

Next, the algorithm checks the following cases one by one in order to detect if there is a constraint that forces the orientation of the edge (v,u):

- If F_{uv} contains an edge that has been oriented away from u, then the algorithm fixes the orientation $(v \rightarrow u)$ (forced by valley-freeness) and orients all edges in H_{uv} towards u (forced by $\prec$).
- If L_{uv} contains an edge that has been oriented towards u, then the algorithm fixes the orientation $(v \rightarrow u)$ (forced by $\prec$) and orients all edges in H_{uv} towards u (again, forced by $\prec$).
- If there is an edge in $L_{uv} \cap F_{uv}$, the algorithm fixes the orientation $(v \rightarrow u)$ and orients all edges in H_{uv} towards u. If (v,u) were oriented differently, all edges in F_{uv} and thus also the edge in L_{uv} would have to be oriented towards u, implying $(v \rightarrow u)$ by the $\prec$ constraints.

If any of these cases applies, the processing of (v,u) is finished. Otherwise, the processing of (v,u) continues as follows: For every pair of edges $(u,w) \in H_{uv}$ and $(u,x) \in F_{uv} - H_{uv}$, the algorithm adds the path $(x\ u\ w)$ to $\mathcal{P}$. (If one of the two edge sets is empty, no paths are added.) Denote the set of these added paths as Q_{uv}. In the example of Figure 3, four paths would be added, namely all paths consisting of one of the two edges (a,u) and (b,u), and one of the two edges (u,e) and (u,f). This completes the processing of (v,u).

After processing all edges (v,u) incident with v in this way, the algorithm now calls itself recursively with parameters $G \setminus v$, $\mathcal{P}'$, and $\prec'$ provided that $V - v$ is not empty. Here, $\mathcal{P}'$ is the set of paths obtained from the paths in $\mathcal{P}$ (after adding the auxiliary paths in the sets Q_{uv}) by restricting them to $V - v$, and $\prec'$ is the restriction of $\prec$ to the nodes in $V - v$. If the recursive call returns 'no', the algorithm outputs 'no'. Otherwise, the algorithm orients the edges not incident with v according to the partial orientation returned by the recursive call, and determines the orientation of each edge (u,v) incident with v whose orientation has not yet been determined as follows:

- If all edges in H_{uv} are directed towards u, the algorithm fixes the orientation $(v \rightarrow u)$.
- If at least one edge in H_{uv} is not directed to u, the algorithm fixes the orientation $(v — u)$. (In this case there cannot be an edge L_{uv} that is oriented towards u, because this would mean that the recursively computed solution for $G \setminus v$ is invalid.)

We remark that the paths Q_{uv} have been added to $\mathcal{P}$ to ensure that either all edges in H_{uv} are oriented towards u or all edges in F_{uv} are oriented towards u. In the latter case, making (v,u) a peer-to-peer edge does not violate valley-freeness and ensures that the $\prec$ constraints at u are satisfied.

Theorem 3.1: The algorithm solves GAO-REXFORD-SIMPLE-CHECK correctly in polynomial time.

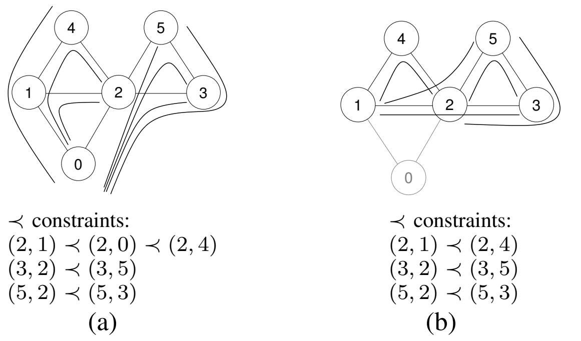


Fig. 4: (a) Instance of GAO-REXFORD-SIMPLE-CHECK constructed from the instance of GAO-REXFORD-CHECK shown in Fig. 2(b). (b) Instance of GAO-REXFORD-SIMPLE-CHECK passed to the recursive call after node 0 has been processed.

Proof: It is clear that the algorithm runs in polynomial time because the number of vertices is reduced by one for each recursive call and the steps before and after the recursive call can be implemented in polynomial time.

The algorithm fixes the orientation of an edge only when the orientation of that edge is implied by constraints and/or the orientations of previously fixed edges, or if the orientation of that edge cannot possibly create any conflicts. Furthermore, the path sets Q_{uv} that are added to $\mathcal{P}$ by the algorithm have the property that they are valley-free in any solution of GAO-REXFORD-SIMPLE-CHECK.

In any valid orientation of G, either all edges of H_{uv} must be directed to u (if (u,v) is oriented as $(v \rightarrow u)$) or all edges of F_{uv} must be directed to u (if (u,v) is oriented as $(v \leftarrow u)$ or $(v — u)$). Therefore, if the algorithm answers 'no' because it wants to orient an edge differently from an orientation that has been fixed earlier, or because it cannot find a vertex that is not traversed by a path and has no incoming edge, it is clear that the given instance is a no-instance. ■

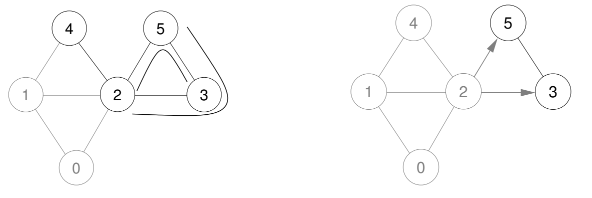
By Lemma 3.1, we obtain the following corollary.

Corollary 3.1: There is a polynomial-time algorithm for problem GAO-REXFORD-CHECK.

We illustrate our algorithm with a complete example. Consider the instance of GAO-REXFORD-CHECK shown in Fig. 2(b). First, the algorithm creates the instance of GAO-REXFORD-SIMPLE-CHECK shown in Fig. 4(a), where the graph G and the set of paths $\mathcal{P}$ are drawn (paths consisting of one edge are omitted) and the precedence constraints $\prec$ are listed (the transitive constraint $(2,1) \prec (2,4)$ is implicit).

In the first step, the algorithm picks node 0 as a node v that is not traversed by any path and has no edge oriented towards it. When it processes edge $(0,1)$, we have $F_{10} = \{(1,2),(1,4)\}$, $L_{10} = \emptyset$ and $H_{10} = \emptyset$. None of the rules that fix orientations of edges apply, and hence the edge $(0,1)$ is left undecided for now.

When the algorithm processes edge $(0,2)$, we have $F_{20} = \{(2,5),(2,3)\}$, $L_{20} = \{(2,4)\}$ and $H_{20} = \{(2,1)\}$. The algorithm adds the auxiliary paths $Q_{10} = \{125, 123\}$ and



≺ constraints:
$(3,2) \prec (3,5)$
$(5,2) \prec (5,3)$
(a)

≺ constraints:
(none)
(b)

Fig. 5: (a) Instance of GAO-REXFORD-SIMPLE-CHECK passed to the recursive call after nodes 0 and 1 have been processed. (b) Instance passed to the recursive call after nodes 0, 1, 2, and 4 have been processed. There are no paths left, and the precedence relation $\prec$ is empty.

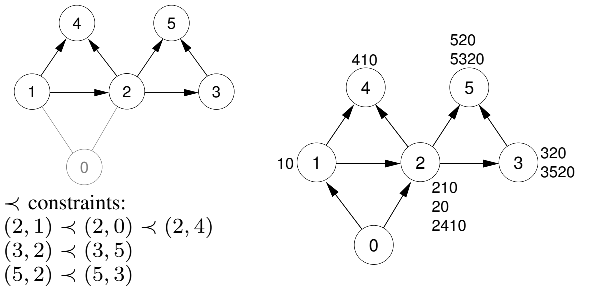
leaves the orientation of $(0, 2)$ undecided for now. The resulting problem instance for the recursive call is now shown in Fig. 4(b), where vertex 0 and its incident edges (which are not passed to the recursive call) are drawn in gray. (Paths of length 1 are omitted as they are not relevant.)

Now, the algorithm picks node 1 as a node v that is not traversed by any path and has no edge oriented towards it. When it processes edge $(1, 4)$, we have $F_{41} = \{(4, 2)\}$, $L_{41} = \emptyset$ and $H_{41} = \emptyset$. When the algorithm processes edge $(1, 2)$, we have $F_{21} = \{(2, 5), (2, 3)\}$, $L_{21} = \{(2, 4)\}$ and $H_{21} = \emptyset$. For none of the two edges $(1, 4)$ and $(1, 2)$, any of the rules to fix orientations apply, so these two edges are left undecided for now. The resulting problem instance for the recursive call is shown in Fig. 5(a), where vertex 1 and its incident edges are also drawn in gray.

Next, the algorithm can pick node 2 or 4, and ties can be broken arbitrarily. Let us assume that the algorithm picks node 2. When it processes the edge $(2, 4)$, we have $F_{42} = H_{42} = L_{42} = \emptyset$. No rules apply, and the orientation of $(2, 4)$ is left undecided for now. When processing $(2, 5)$, we have $F_{52} = \{(5, 3)\}$, $L_{52} = \{(5, 3)\}$ and $H_{52} = \emptyset$. As $F_{52} \cap L_{52} \neq \emptyset$, the algorithm fixes the orientation of $(2, 5)$ as $(2 \to 5)$. The processing of $(2, 3)$ is analogous and fixes the orientation of $(2, 3)$ as $(2 \to 3)$. The resulting problem instance for the next nontrivial recursive call (skipping the processing of node 4 that has no effect as that node does not have any incident edges left) is shown in Fig. 5(b).

Next the algorithm can pick 3 or 5, ties can be broken arbitrarily. Let us assume the algorithm picks node 3. The processing of edge $(3, 5)$ does not fix any orientation. The graph obtained from deleting node 3 contains only node 5 and no edges. The recursive call for this graph terminates the recursion and returns the graph with only node 5 as a valid partial orientation.

As the recursive calls terminate, the nodes are processed in reverse order, and each node decides the orientations of the edges to nodes that have already been processed during this second phase of the algorithm. When node 3 is processed, the algorithm fixes the orientation of $(3, 5)$ as $(3 \to 5)$ since



≺ constraints:
$(2,1) \prec (2,0) \prec (2,4)$
$(3,2) \prec (3,5)$
$(5,2) \prec (5,3)$
(a)

(b)

Fig. 6: (a) Situation before node 0 is processed during the tail end of the recursion. (b) Solution obtained for the instance of GAO-REXFORD-CHECK of Fig. 2(b).

H_{53} is empty. When node 4 is processed, nothing happens. When node 2 is processed, edge $(2, 4)$ is oriented as $(2 \to 4)$ since H_{24} is empty. Now, node 1 is processed. Edge $(1, 4)$ is oriented as $(1 \to 4)$ since H_{41} is empty, and edge $(1, 2)$ is oriented as $(1 \to 2)$ since H_{21} is empty. The situation is now as shown in Fig. 6(a).

When node 0 is processed, edge $(0, 1)$ is oriented as $(0 \to 1)$ since $H_{10} = \emptyset$. Edge $(0, 2)$ is oriented as $(0 \to 2)$ since all edges in $H_{20} = \{(2, 1)\}$ have been oriented towards 2. The final solution obtained for the original instance of GAO-REXFORD-CHECK is shown in Fig. 6(b). Since the algorithm has found a valid partial orientation, it answers ‘yes’ for the given instance of GAO-REXFORD-CHECK. This concludes the detailed example.

We now show that an analogous result to Corollary 3.1 holds in the more realistic scenario where BGP policies are represented according to the SSPP model. We adapt our approach to address the problem GAO-REXFORD-CHECK-SUCCINCT. Given an instance $\tilde{S} = (G, \tilde{\mathcal{P}}, \tilde{\Lambda})$ of the problem, we can transform it into an instance $S' = (G, \mathcal{P}, \prec)$ of problem GAO-REXFORD-SIMPLE-CHECK as follows: We let $\mathcal{P}$ be the union of the sets $\tilde{\mathcal{P}}^u$ of path fragments for any u, and we define that $uw \prec uv$ if $\tilde{\mathcal{P}}^u$ contains path fragments $p_1 = (u\ w\ x)$ and $p_2 = (u\ v\ y)$, for some nodes x, y, with $\tilde{\lambda}^u(p_1) < \tilde{\lambda}^u(p_2)$. Furthermore, we make the relation $\prec$ transitive by determining the transitive closure. Analogously to Lemma 3.1, one can show that a partial orientation of G is a solution to $\tilde{S}$ if and only if it is a solution to S'. Thus, a polynomial-time algorithm for problem GAO-REXFORD-SIMPLE-CHECK gives a polynomial-time algorithm for problem GAO-REXFORD-CHECK-SUCCINCT. By Theorem 3.1, we obtain the following corollary.

Corollary 3.2: There is a polynomial-time algorithm for problem GAO-REXFORD-CHECK-SUCCINCT.

IV. FINDING AN ORIENTATION IS HARD IF PEERS ARE PREFERRED TO PROVIDERS

In this section we show that changing condition *prefer-customer* so that peers are ranked better than providers makes the check NP-hard. We call Gao-Rexford* this variant and prefer-customer* the corresponding condition. This result implies that an algorithm to check compliance with the Gao-Rexford* conditions would be of little use, given that it would fail to be either correct, or complete, or efficient and further justifies the original Gao-Rexford model where routes from peers and routes from providers are ranked into the same class. Namely, we consider the following problem:

Problem:	GAO-REXFORD-STRICT-CHECK
Instance:	An instance $S = (G, \mathcal{P}, \Lambda)$ of SPP.
Question:	Can G be partially oriented to a customer-provider graph such that S satisfies the Gao-Rexford* conditions?

We show that GAO-REXFORD-STRICT-CHECK is NP-hard by reducing the 3SAT problem to it. In the 3SAT problem you are given a set of clauses, each consisting of three literals, and are asked to find a truth assignment to the Boolean variables such that each clause has at least one true literal. Given an instance φ of 3SAT, we describe how to build a corresponding instance $S_\varphi = (G, \mathcal{P}, \Lambda)$ of SPP such that G admits a partial orientation to a customer-provider graph satisfying the Gao-Rexford* conditions if and only if φ admits a solution.

We build the SPP instance S_φ by composing some special gadgets. The goal is to introduce for each clause a clause gadget which forces the presence of a cycle in the customer-provider graph whenever the clause is not satisfied.

First of all, consider the *symmetric configuration* $\mathcal{SC}(u, v, w)$ represented in Fig. 7. In this configuration vertex v is reached by the two paths P_1 and P_2 received from its neighbor u and by the two paths P_3 and P_4 received from its neighbor w. The ranking function λ^v at vertex v is such that $P_1 < P_3 < P_4 < P_2$, while λ^u and λ^w are such that these four paths are ranked better than any other paths.

Lemma 4.1: Let $S_\varphi = (G, \mathcal{P}, \Lambda)$ be an instance of SPP containing a symmetric configuration $\mathcal{SC}(u, v, w)$. In any solution of GAO-REXFORD-STRICT-CHECK, one of the following holds: *(i)* v is a customer of both u and w; *(ii)* v is a provider of both u and w; or *(iii)* v is a peer of both u and w.

Proof: Since v prefers P_1 to P_3, the orientations $(u \leftarrow v \leftarrow w)$, $(u \leftarrow v — w)$, and $(u — v \leftarrow w)$ make λ^v violate the prefer-customer* condition. Symmetrically, since v prefers P_4 to P_2, the orientations $(u \rightarrow v \rightarrow w)$, $(u — v \rightarrow w)$, and $(u \rightarrow v — w)$ make λ^v violate the prefer-customer* condition. Hence, the remaining three orientations $(u \rightarrow v \leftarrow u)$, $(u \leftarrow v \rightarrow u)$, and $(u — v — u)$ are the only possible. Observe that in any feasible orientation all edges between u and 0 along P_1 and P_2 can be oriented away from 0. The same holds for edges between w and 0 along P_3 and P_4. ■

By simply concatenating two symmetric configurations $\mathcal{SC}(u, v, w)$ and $\mathcal{SC}(v, w, x)$ one obtains a configuration where either all the three edges are peer-to-peer edges, or edge $e_1 = (u, v)$ has the same orientation of edge $e_2 = (w, x)$. We call this configuration, denoted $\mathcal{DC}(e_1, e_2)$ and depicted in Fig 8(a), *de-coupling configuration*, since in any solution of GAO-REXFORD-STRICT-CHECK a directed cycle can not traverse it.

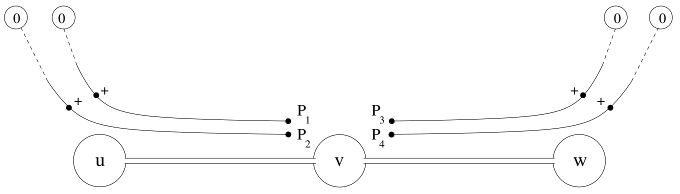


Fig. 7: The *symmetric configuration* $\mathcal{SC}(u, v, w)$. All the paths originate from 0. Small dots along the paths represent entries in $\mathcal{P}$. Small + signs near the dots mean that the corresponding paths are preferred (assigned lower values of λ) by the corresponding vertex. Function λ^v ranks $P_1 < P_3 < P_4 < P_2$.

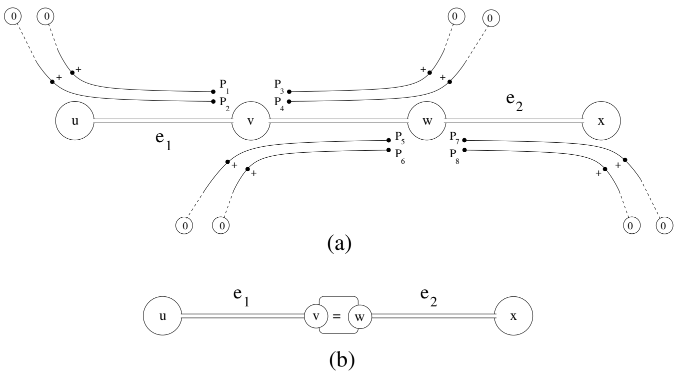


Fig. 8: (a) The *de-coupling configuration* is composed by two symmetric configurations $\mathcal{SC}(u, v, w)$ and $\mathcal{SC}(v, w, x)$. We compactly represent the de-coupling configuration as in (b).

For each Boolean variable x_i of φ we introduce a *variable gadget* (see Fig. 9) that is composed by a de-coupling configuration $\mathcal{DC}(e_1, e_2)$ plus a *true-false path* $(v\ u\ \dots\ x\ w\ \dots\ 0)$ as depicted in the figure.

Lemma 4.2: Let $S_\varphi = (G, \mathcal{P}, \Lambda)$ be an instance of SPP containing a variable gadget. In any solution of GAO-REXFORD-STRICT-CHECK, the edges of the true-false path are either all directed from x to u or from u to x.

Proof: Due to Lemma 4.1, the Gao-Rexford* conditions can only be satisfied by one of the following orientations of edges e_1, e_2: *(i)* $(u \rightarrow v)$ and $(w \rightarrow x)$; *(ii)* $(u \leftarrow v)$ and $(w \leftarrow$

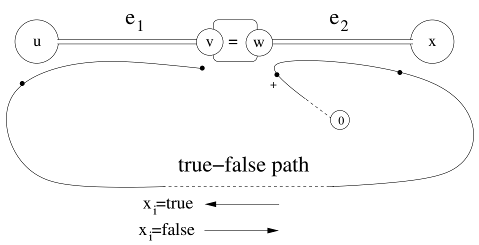


Fig. 9: The *variable gadget* is composed by a de-coupling configuration plus the *true-false path* depicted in the figure. All the edges of the true-false path are either directed from x to u (the variable is true) or from u to x (the variable is false).

x); or (*iii*) both e_1 and e_2 are peer-to-peer edges. Case (*iii*) is ruled out, because it would violate valley-freeness of the true-false path. Hence, Cases (*i*) and (*ii*) are the only two possible. Also, in any feasible orientation all edges between 0 and w along the true-false path can be oriented away from 0. ■

Intuitively, the orientation from x to u of the true-false path of variable x_i will be associated with a true value for x_i, while the opposite orientation from u to x will be associated with a false value. These orientations are exploited by the tap configurations, which are responsible for "extracting" truth values from a true-false path.

Let $C = l_1 \vee l_2 \vee l_3$ be a clause of φ, and let x_1, x_2, and x_3 be its variables. For each l_i, $i = 1, 2, 3$, we introduce a *tap configuration* $\mathcal{TC}(u, v)$, shown in Figs. 10 and 11. A tap configuration $\mathcal{TC}(u, v)$ consists of a de-coupling configuration and a pair of adjacent vertices (u, v). The de-coupling configuration is attached on one side to vertex u and, on the other side, to the true-false path of the variable gadget of x_i. The attachment to the true-false path depends on the literal l_i. In particular, if l_i is a direct literal, then we attach to the true-false path the tap configuration in Fig. 10. Otherwise, if l_i is a negated literal, we attach to the true-false path the tap configuration in Fig. 11. The function of tap configurations is to force edge (u, v) to be oriented from v to u in any partial orientation of G that satisfies the Gao-Rexford* conditions, whenever the corresponding literal l_i is false (i.e., if $l_i = x_i$ and $x_i =$ false, or if $l_i = \bar{x}_i$ and $x_i =$ true).

With arguments similar to the ones used in the proofs of Lemmas 4.1-2, the following lemma can be easily proved.

Lemma 4.3: Let $S_\varphi = (G, \mathcal{P}, \Lambda)$ be an instance of SPP containing a tap configuration $\mathcal{TC}(u, v)$ attached to the true-false path of variable x_i and corresponding to a direct (negated) literal of x_i. In any solution of GAO-REXFORD-STRICT-CHECK, if the edges of the true-false path are oriented towards 0 (away from 0, respectively), then edge (u, v) is oriented from v to u.

Before describing the clause gadget, we need to introduce a final tool. Given two non-adjacent vertices u and v of G, the purpose of the *forcing configuration*, denoted $\mathcal{FC}(u, v)$ and depicted in Fig. 12, is to add to G an edge (u, v) that is forced to be oriented from u to v in any solution of GAO-REXFORD-STRICT-CHECK. To this purpose, we add two paths $P_1 = (0 \ldots u\ v\ w\ x)$ and $P_2 = (0 \ldots u\ v\ x\ w)$ to $\mathcal{P}$. Paths P_1 and P_2 are assigned highest ranks by λ^u and λ^v. Also, P_1 is best ranked by λ^w and P_2 is best ranked by λ^x (see Fig. 12).

Lemma 4.4: Let $S_\varphi = (G, \mathcal{P}, \Lambda)$ be an instance of SPP containing a forcing configuration $\mathcal{FC}(u, v)$. In any solution of GAO-REXFORD-STRICT-CHECK, edge (u, v) is oriented from u to v.

Proof: Consider the three possible orientations for edge (w, x). If $(w \rightarrow x)$ or $(w \leftarrow x)$ then, for both P_1 and P_2 to be valley-free, it must be $(u \rightarrow v)$. Similarly, if $(w — x)$ then, for P_1 and P_2 to be valley-free it must also be $(u \rightarrow v)$.

Hence, suppose $(u \rightarrow v)$. Orient the edges $(w \rightarrow x)$, $(v \rightarrow w)$, $(v \rightarrow x)$, and orient all the other edges of P_1 and P_2 from 0 to u. It is easy to check that P_1 and P_2 are valley-free, that v, w, and x do not form a directed cycle, and that, whatever the other paths traversing u and v are, the ranking of paths P_1 and P_2 makes all the vertices of $\mathcal{FC}(u, v)$ compliant with the prefer-customer* condition. ■

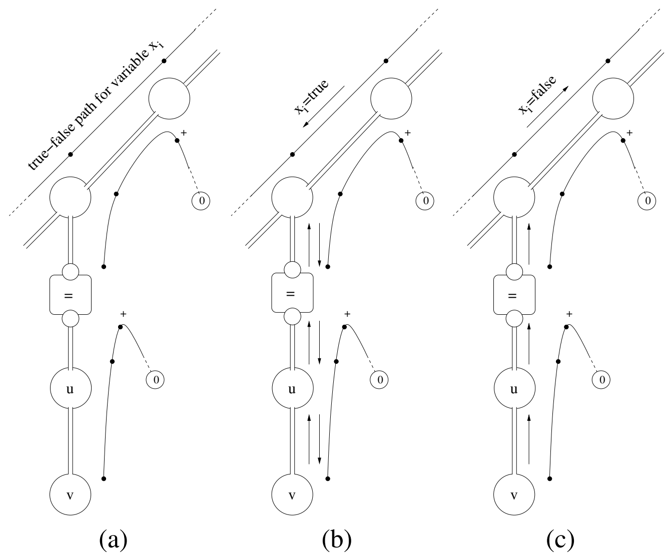


Fig. 10: (a) The *tap configuration* $\mathcal{TC}(u, v)$ for a direct literal x_i. In any partial orientation satisfying the Gao-Rexford* conditions, if $x_i =$ true, edge (u, v) may have an arbitrary orientation (b). If $x_i =$ false, then v is a customer of u (c).

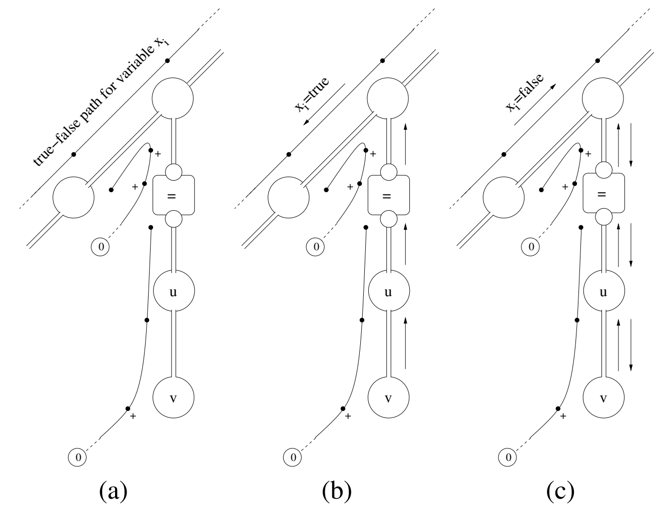


Fig. 11: The *tap configuration* $\mathcal{TC}(u, v)$ for a negated literal $\bar{x}_i$. In any partial orientation satisfying the Gao-Rexford* conditions, if $x_i =$ true, then v is a customer of u (b). If $x_i =$ false, edge (u, v) may have an arbitrary orientation (c).

The *clause gadget* for a clause $C = l_1 \vee l_2 \vee l_3$ is shown in Fig. 13. It consists of three tap configurations $\mathcal{TC}(u_1, v_1)$, $\mathcal{TC}(u_2, v_2)$, and $\mathcal{TC}(u_3, v_3)$ (direct or negated, depending on the case) attached to the true-false paths of the variable gadgets for x_1, x_2, and x_3, respectively. Further, we add three forcing

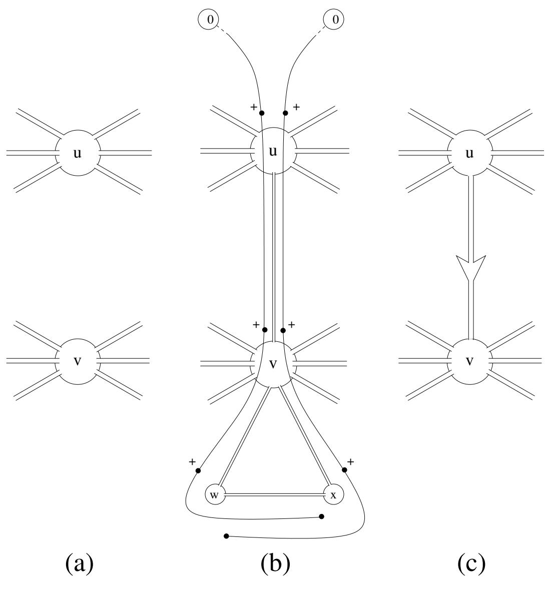


Fig. 12: The forcing configuration $\mathcal{FC}(u, v)$. Let u and v be two non-adjacent vertices (a). Whatever the ranking functions at vertices u and v are, the two most preferred paths added to u and v (b) force the new edge (u, v) to be oriented from u to v. We compactly depict the forcing gadget as in (c).

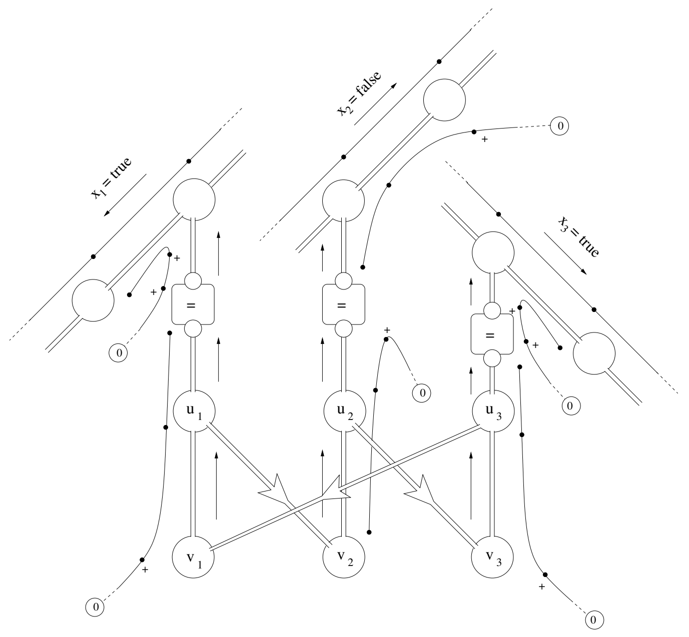


Fig. 13: The clause gadget for clause $\bar{x}_1 \vee x_2 \vee \bar{x}_3$. There is a directed cycle of six edges $(u_1\ v_2\ u_2\ v_3\ u_3\ v_1\ u_1)$ when $x_1 =$ true, $x_2 =$ false, and $x_3 =$ true.

configurations $\mathcal{FC}(u_1, v_2)$, $\mathcal{FC}(u_2, v_3)$, and $\mathcal{FC}(u_3, v_1)$.

Lemma 4.5: Let $S_\varphi = (G, \mathcal{P}, \Lambda)$ be an instance of SPP corresponding to the 3SAT instance φ. Graph G admits a partial orientation to a customer-provider graph satisfying the Gao-Rexford* conditions if and only if φ admits a solution.

Proof: Suppose that φ admits a solution. Consider the true-false-path associated with each variable and orient its edges according to the truth value satisfying φ. Each clause has at least a true literal. Consider a clause $C = l_1 \vee l_2 \vee l_3$ and the corresponding clause gadget containing the tap configurations $\mathcal{TC}(u_1, v_1)$, $\mathcal{TC}(u_2, v_2)$, and $\mathcal{TC}(u_3, v_3)$. Suppose, without loss of generality, that C has the true literal l_2. Orient edges $(u_1 \leftarrow v_1)$, $(u_2 \rightarrow v_2)$, and $(u_3 \leftarrow v_3)$. All the remaining edges are oriented as described above for each gadget. Observe that, in this way, there cannot be any directed cycles within a single clause gadget. Due to the de-coupling configurations inserted into the tap configurations, there cannot be a directed cycle involving two clause gadgets either. Finally, due to the de-coupling configuration into the variable gadget, there cannot be a directed cycle using the true-false path.

Conversely, suppose that G admits partial orientation to a customer-provider graph satisfying the Gao-Rexford* conditions. Consider the clause gadget corresponding to a clause $C = l_1 \vee l_2 \vee l_3$ and containing the tap configurations $\mathcal{TC}(u_1, v_1)$, $\mathcal{TC}(u_2, v_2)$, and $\mathcal{TC}(u_3, v_3)$. For the orientation to satisfy the Gao-Rexford* conditions, at least one edge (u_i, v_i), $i = 1, 2, 3$ must be oriented $(u_i \rightarrow v_i)$. By Lemma 4.3, this implies that the true-false path for the corresponding variable is oriented so that literal l_i is true, satisfying clause C. ■

Theorem 4.1: Problem GAO-REXFORD-STRICT-CHECK is NP-hard.

Proof: We exploit the reduction from 3SAT to GAO-REXFORD-STRICT-CHECK described above. Let $S_\varphi = (G, \mathcal{P}, \Lambda)$ be the instance corresponding to instance φ of 3SAT. By Lemma 4.5, we have that G admits a partial orientation to a customer-provider graph satisfying the Gao-Rexford* conditions, if and only if φ admits a solution. Since S_φ can be constructed in polynomial time, the statement follows. ■

We remark that a construction similar to the one described in this section allows to prove an analogous result in the SSPP model. Namely, we consider the following problem:

Problem:	GAO-REXFORD-STRICT-CHECK-SUCCINCT
Instance:	An instance $\tilde{S} = (G, \tilde{\mathcal{P}}, \tilde{\Lambda})$ of SSPP.
Question:	Can G be partially oriented to a customer-provider graph such that $\tilde{S}$ satisfies the Gao-Rexford* conditions?

Theorem 4.2: Problem GAO-REXFORD-STRICT-CHECK-SUCCINCT is NP-hard.

Proof: We prove this by showing that the same filtering and ranking policies adopted in the gadgets described for the reduction from 3SAT to GAO-REXFORD-STRICT-CHECK can be applied by just using the path fragments of the SSPP model.

First of all, given the instance $S_\varphi = (G = (V, E), \mathcal{P}, \Lambda)$, each path $P \in \mathcal{P}$ can be suitably replaced by fragments such that their chain produces $\mathcal{P}$. It can be verified that this easily applies to the symmetric, de-coupling, and forcing configurations, as well as to the variable gadget. For the tap gadget, we need to apply a small variation. Consider Fig. 14, that shows how the tap gadget is attached to the variable

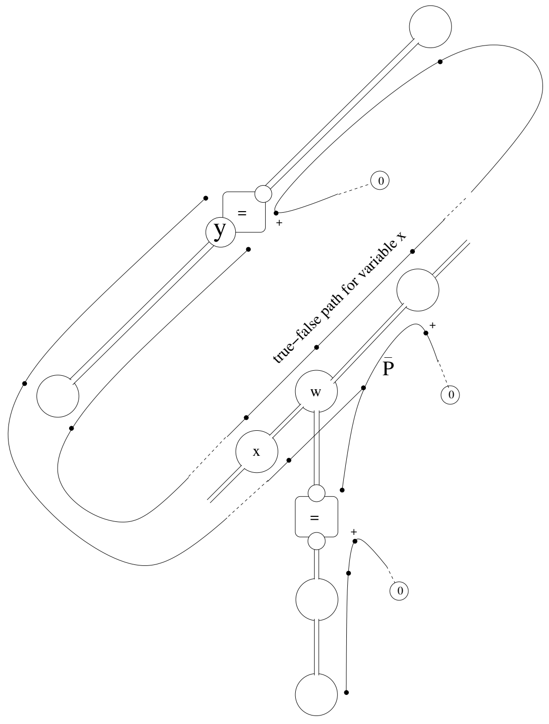


Fig. 14: To show that GAO-REXFORD-STRICT-CHECK is NP-hard, paths of the tap gadget must be slightly modified when combined with the variable gadget.

gadget. By looking at the figure, we can see that there is no combination of fragments that allows vertex w to propagate the true-false path to its neighbor x while filtering out path $\bar{P}$. However, even if path $\bar{P}$ is propagated as far as vertex y, the gadget continues to work as required.

About the ranking, all the preferences represented by $+$ signs can be expressed using per-neighbor rankings, which are compatible with the SSPP model. The only exception is the symmetric configuration in Fig. 7, where paths P_1, P_2, P_3, and P_4 cannot be ranked on a per-neighbor basis. However, recall that the purpose of the symmetric configuration is to ensure that neighbors u and w of v are of the same type (both customers, both providers, or both peers). The same constraint can be rendered in the SSPP model by requiring that $\tilde{\lambda}^u((v\ u\ x)) = \tilde{\lambda}^u((v\ w\ y))$ for every x, y such that $(u, x) \in E$ and $(w, y) \in E$. ■

V. CONCLUSIONS

A stable BGP routing is a crucial requirement for a proper operation of the Internet. Lots of research efforts have therefore been devoted to finding BGP configuration guidelines that satisfy this requirement. To date, the conditions stated by Gao and Rexford [1] are the most widely accepted to ensure guaranteed routing stability, and indeed are regarded as a fundamental reason for the stability of the Internet.

Due to the relevance of these conditions, the feasibility of checking whether a BGP configuration honors them has a remarkable practical importance. We show an efficient algorithm that, given a BGP configuration, checks whether there exists an assignment of peer-peer and customer-provider relationships that complies with the Gao-Rexford conditions. We also show that a slight modification of the Gao-Rexford conditions, which makes them more realistic, is sufficient to make the problem intractable. Our results also hold in a model where BGP policies are expressed using lifelike constructions such as "prefer routes through customer x over those through customer y" or "pass routes learned from neighbor w to neighbor v".

Considering policy guidelines from this algorithmic perspective opens several interesting problems. First of all, one could understand how complex it is to check if a BGP configuration complies with guidelines that support backup routing [10]. Also, the complexity of checking less constraining conditions for guaranteed BGP convergence [2] is still unknown. All of the above questions, as well as the problems we have addressed in this paper, can also be reconsidered in different models of BGP policies (see, e.g., [20]).

REFERENCES

[1] L. Gao and J. Rexford, "Stable Internet routing without global coordination," in *Proc. SIGMETRICS*, 2000.
[2] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *Transactions on Networking*, vol. 10, no. 2, 2002.
[3] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," IETF RFC 4271, 2006.
[4] A. Fabrikant and C. Papadimitriou, "The complexity of game dynamics: BGP oscillations, sink equilibria, and beyond," in *Proc. SODA*, 2008.
[5] T. G. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *Proc. SIGCOMM*, 1999.
[6] T. G. Griffin and G. T. Wilfong, "A Safe Path Vector Protocol," in *Proc. INFOCOM*, 2000.
[7] C. T. Ee, V. Ramachandran, B.-G. Chun, K. Lakshminarayanan, and S. Shenker, "Resolving inter-domain policy disputes," in *Proc. SIGCOMM*, 2007.
[8] N. Feamster, R. Johari, and H. Balakrishnan, "Implications of autonomy for the expressiveness of policy routing," *Transactions on Networking*, vol. 15, no. 6, 2007.
[9] A. D. Jaggard and V. Ramachandran, "Robustness of class-based path-vector systems," in *Proc. ICNP*, 2004.
[10] L. Gao, T. Griffin, and J. Rexford, "Inherently safe backup routing with BGP," in *Proc. INFOCOM*, 2001.
[11] R. Sami, M. Schapira, and A. Zohar, "Searching for stability in inter-domain routing," in *Proc. INFOCOM*, 2009.
[12] L. Gao, "On inferring Autonomous System relationships in the Internet," *Transactions on Networking*, vol. 9, no. 6, 2001.
[13] G. Di Battista, T. Erlebach, A. Hall, M. Patrignani, M. Pizzonia, and T. Schank, "Computing the types of the relationships between Autonomous Systems," *Trans. on Networking*, vol. 15, no. 2, 2007.
[14] S. Kosub, M. G. Maaß, and H. Täubig, "Acyclic type-of-relationship problems on the Internet," in *Proc. CAAN*, 2006.
[15] S. Epstein, K. Mattar, and I. Matta, "Principles of safe policy routing dynamics," in *Proc. ICNP*, 2009.
[16] B. Quoitin and S. Uhlig, "Modeling the routing of an Autonomous System with C-BGP," *IEEE Network*, vol. 19, no. 6, 2005.
[17] P. Traina, D. McPherson, and J. Scudder, "Autonomous System Confederations," IETF RFC 5065, 2007.
[18] "The Internet Routing Registry: History and Purpose," `http://www.ripe.net/db/irr.html`.
[19] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "Policy disputes in path-vector protocols," in *Proc. ICNP*, 1999.
[20] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *Proc. SIGCOMM*, 2005.