Computing with BGP: from Routing Configurations to Turing Machines

Marco Chiesa⋆ Luca Cittadini⋆ Giuseppe Di Battista⋆ Laurent Vanbever† Stefano Vissicchio†
⋆ Dept. of Computer Science and Automation, Roma Tre University † Université Catholique de Louvain
⋆{chiesa,ratm,gdb}@dia.uniroma3.it †{firstname.lastname}@uclouvain.be

***Abstract*—Because of its practical relevance, the Border Gateway Protocol (BGP) has been the target of a huge research and industrial effort since more than a decade and a BGP routing theory has been developed out of that effort.**

In this paper, we show that there exists a mapping between BGP and a logic circuit. We show simple networks with routers with elementary BGP configurations that simulate logic gates, clocks and flip-flops, and we show how to interconnect them to simulate arbitrary logic circuits. We then investigate the implications of such a mapping on the computational complexity of BGP problems. We show that, under reasonable assumptions on message timings, BGP has the same computing power as a Turing Machine. As a consequence, we devise a new method for studying the complexity of analyzing BGP configurations and exploit such a method to give several new complexity bounds. Also, if message timings are unrestricted, BGP can simulate a combinational logic circuit, which allows us to prove the NP-hardness of a new variant of a well-known BGP problem.

Finally, we investigate whether the mapping is still feasible when BGP policies are restricted, e.g., in iBGP or when Local Transit Policies or Gao-Rexford conditions are enforced.

I. Introduction

The Border Gateway Protocol (BGP) [1] is the de-facto standard protocol that regulates inter-domain routing. Each Autonomous System (AS) willing to join the Internet has to configure both eBGP and iBGP on its routers. Routing information about Internet destinations is exchanged via eBGP, while iBGP distributes this information inside the same AS.

BGP has been designed to allow each AS to specify its own routing policies in complete autonomy, in such a way that the AS can fully control routes that it accepts, prefers, and propagates. Such a rich policy expressiveness has been shown to be the root cause of routing and forwarding anomalies that can occur in both eBGP [2] and iBGP [3].

Because of its practical relevance for Internet operationand its lack of correctness guarantees, BGP issues have been the focus of a huge research and industrial effort in the last 15 years. Results of such an effort encompass experimental measurements of disruptions due to BGP (e.g., [4], [5]), formal analyses of the protocol (e.g., [2], [3]), proposal of configuration guidelines (e.g., [6]) and of protocol modifications (e.g., [7]), and practical approaches to check a given configuration for correctness (e.g., [8], [9]).

In this paper, we unveil a mapping between eBGP configurations and elementary logic gates. By fully developing this intuition, we show that eBGP is also powerful enough to simulate memory and clock components, and encode arbitrary logic circuits. We build the mapping assuming a simplified model for BGP routing policies which does not capture advanced BGP features like MED or conditional advertisement.

We investigate the theoretical consequences of the mapping on the computational complexity of several BGP problems, using two different models for BGP dynamics.

First, we consider a model where some constraints are imposed on BGP message timings. In particular, we assume that each link has a minimum and a maximum delay. We show that eBGP configurations can simulate arbitrary Turing Machines in this model. Two implications derive from this. On one hand, we have that policy-based protocols like BGP intrinsically have the same computational power of Turing Machines, even when simple policies are considered. On the other hand, BGP routing problems, like convergence and route propagation, can be shown to be PSPACE-hard in the considered model. This implies that such problems cannot be solved by a SAT-solver [10].

Second, we consider a purely asynchronous model in which BGP messages can be arbitrarily (even if not indefinitely) delayed. Hence, we revisit previous work by devising a proving method based on the mapping between BGP configurations and combinational logic circuit. We show how to apply this method to both known BGP problems and their variants.

Finally, we investigate the impact of policy restrictions on the complexity of BGP problems. We analyze both iBGP networks and eBGP policy configuration paradigms like the well-known Gao-Rexford conditions [6] and the widely used Local Transit Policies [11]. Unfortunately, we find that problems remain hard in all those settings. Some robustness problems remain hard even when convergence to a single stable state is guaranteed. Our findings extend the results claimed in [12], and define the border with open problems more precisely.

The rest of the paper is organized as follows. Section II defines the mapping between BGP configurations and logic circuits. Section III shows the implications of the mapping on the complexity of BGP problems in different models. Section IV uses the mapping to analyze the impact of policy restrictions to our complexity results. Section V discusses the related work, and Section VI concludes the paper.

II. BGP Configurations as Logic Circuits

BGP's most prominent feature is the support for routing policies that each BGP router can autonomously define. Routing policies are used to specify which routes should be accepted from (or announced to) which neighbors, and to assign different degrees of preference to different routes.

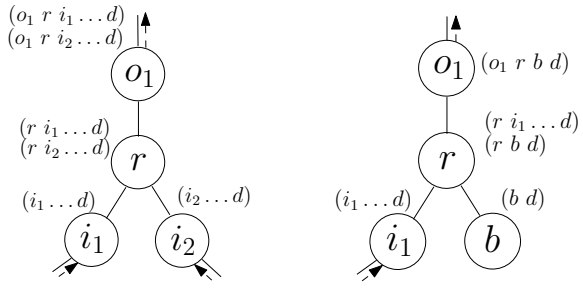
In this paper we rely on the well-known SPP formalism [2] to model eBGP configurations (in Section IV-A we use a similar model to represent iBGP configurations). In SPP, an eBGP configuration is represented as a graph where every node is an Autonomous System (AS) and every edge is an eBGP peering. Since BGP routers treats different destinations separately, we focus on one destination at the time. The destination is represented by a special node d, to which all other nodes try to establish a route. A route is a simple path on the graph. Each node can specify its own policy, which is modeled as the totally ordered set of all the routes that the node accepts towards the destination. SPP encodes the BGP decision process directly in the configuration of the policies in such a way that nodes can use an extremely simple algorithm to select their best route [2].

Despite the fact that SPP is a simplified model for BGP policies (it does not capture MEDs and conditional route announcements, just to name a few), in this section we show that SPP instances representing eBGP configurations can emulate any logic circuit. First, we show eBGP configurations that implement elementary logic gates. Second, we describe eBGP configurations for more advanced circuital components, namely flip-flops and clocks. Finally, we describe how to arbitrarily connect circuital components.

A. Elementary Logic Gates with eBGP

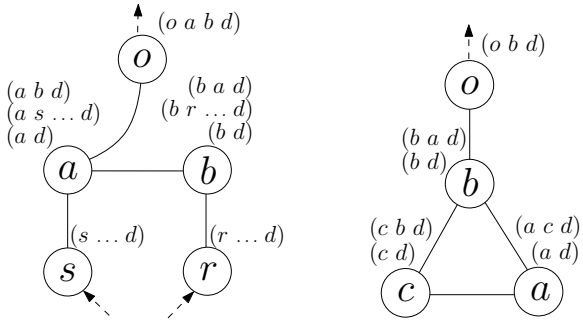
We now show how to build an eBGP configuration that simulates the OR and NOT logic gates. We map the inputs (outputs, resp.) of a logic gate to a set of *input nodes* (*output nodes*, resp.) of the SPP instance. Also, we map the availability of a route to a 1 and the absence of a route to a 0. In particular, the availability (absence, resp.) of a route at an output node r at time t means that the logic gate's output signal at t is 1 (0, resp.).

The eBGP configurations simulating the OR and the NOT logic gates are shown in Fig. 1. The graphical convention we use in the figure is adopted throughout the paper, unless differently specified. ASes are represented by circles, and solid edges represent eBGP peerings. A list of paths is specified beside each AS. Each list contains the paths that the AS accepts (i.e., paths that are not filtered out by the routing policy) in a descending order of preference. All routes refer to the same destination d. We use dots inside a path when we do not specify the entire path, so $(a\ b \dots d)$ represents a path that start at a, traverses b and ends at d. Incoming and outgoing dashed arrows indicate input and output nodes, respectively. For the sake of brevity, whenever it is clear from the context, we omit node d and its peerings. For example, in Fig. 1(b), d should be considered directly attached to b. Note that b is guaranteed to have a route to d, since d announces its presence to every neighbor and b accepts path $(b\ d)$.



(a) The OR gate. (b) The NOT gate.

Fig. 1. eBGP networks simulating basic logic gates.



(a) A DISAGREE with two extra nodes S and R behaves as a flip-flop.

(b) A BAD-GADGET plus output node o behaves as a clock.

Fig. 2. eBGP networks simulating memory and clock.

Fig. 1(a) represents an eBGP configuration corresponding to the OR gate. Since node o_1 only accepts routes from r, o_1 will have a route to d if and only if at least one between i_1 and i_2 has a route to d. Similarly, Fig. 1(b) represents an eBGP configuration simulating the NOT gate. In this configuration, o_1 has a route to d if and only if i_1 has no route to d. Indeed, if i_1 has a route to d, r receives and selects the route from i_1 because of its preferences. Thus, o_1 will end up with no route, since o_1 does not accept path $(o_1\ r\ i_1 \dots d)$, as shown by the absence of the path in the list aside o_1 in Fig. 1(b)). On the contrary, if i_1 has no route to d, then r selects $(r\ b\ d)$ and, consequently, o_1 selects $(o_1\ r\ b\ d)$.

B. Memory and Clock with Popular eBGP Gadgets

Besides encoding elementary gates, eBGP is powerful enough to simulate more complex logic components, like flip-flops and clock generators.

Fig. 2(a) shows an eBGP configuration that simulates an SR flip-flop. This flip-flop has two inputs S (set bit) and R (reset bit) and one output Q. The flip-flop stores and outputs a 1 (0, resp.) whenever the set (resp., reset) bit is set to 1 (0, resp.). If both set and reset bits are set to 0, then output Q is the stored value. Setting both S and R to 1 is not allowed.

The configuration in Fig. 2(a) simulates this behavior. It is based on the presence of a well-known BGP gadget, called DISAGREE [2], that has two stable states. Indeed, nodes a and

b form a DISAGREE. In one stable state, nodes a and b select paths $(a\ b\ d)$ and $(b\ d)$, respectively. In the other one, nodes a and b select $(a\ d)$ and $(b\ a\ d)$, respectively. Depending on whether nodes s and r receive a route, we have the following three cases. If s announces a route to a, then a never selects $(a\ d)$, since path $(a\ s \dots d)$ is available and more preferred than $(a\ d)$. Hence, b has to select $(b\ d)$ and a can select its best path $(a\ b\ d)$. Symmetrically, if r announces a route to b, then a has to select $(a\ d)$. Finally, if neither s nor r receives a route, then the DISAGREE does not change its stable state. As a consequence, node o has an available path to d if and only if node a selects path $(a\ b\ d)$, hence mirroring the output of an SR flip-flop.

Further, the dynamics of eBGP configuration that admits no stable state are conceptually similar to those of clock generators. A *clock generator* is a logic circuit producing a signal that oscillates between 1 and 0. The BAD-GADGET [2], shown in Fig. 2(b), is a gadget that never converges to a stable state. It consists in a cycle of three nodes a, b, and c, in which each node prefers a route through its successor instead of a direct route to d. When the gadget oscillates, node a alternatively selects paths $(a\ c\ d)$ and $(a\ d)$. Since o does not accept path $(a\ c\ d)$ from a, o has a route only when node a selects $(a\ d)$. Therefore, the output node o will alternate between having a route and not having any route, as for a clock generator. Observe that the clock of Fig. 2(b) can be thought in terms of the circular interconnection of 3 NOT gates of Fig. 1(b).

C. Simulating Arbitrary Logic Circuits

Now that we have the elementary logic components, it would be tempting to simply interconnect them using eBGP peerings. Such an operation is needed for building: (i) the AND gate, using OR and NOT and applying the De Morgan's laws; (ii) arbitrary logic gates as a combination of AND, OR, and NOT; and (iii) arbitrary logic circuits starting from logic gates, flip-flops and clocks. Unfortunately, arbitrary interconnections are not straightforward, especially because of BGP peculiarities.

The first problem we face is that signal propagation in logic circuits has a direction, while routes may traverse an eBGP peering in both ways. We need to prevent routes from being propagated in unintended directions, e.g., "signals" traversing the gates from their output to their input. This can be accomplished by using eBGP policies to accept only routes in the intended direction.

A second and more subtle problem arises with loops. BGP has a built-in control plane loop prevention mechanism [1] which mandates ASes to discard routes containing their own identifier. Because of this mechanism, we need an additional building block to be able to simulate logic circuits where the signal is propagated through a loop. In particular, we interpose a special gadget, called HUB gadget, between any pair of interconnected logic components

The HUB gadget is represented in Fig. 3. Intuitively, it takes a route at its input node i and generates a new, completely

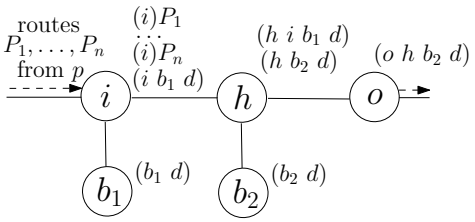


Fig. 3. The HUB gadget we use to interconnect logic components.

different route at its output node o. It can be seen as the concatenation of two NOT gadgets, in which the first NOT gadget filters out the original route and the second NOT gadget generates the new one. No route is produced in output if i receives no external route. In other words, the HUB gadget is able to correctly propagate both the presence of a route (a binary 1) and the absence thereof (a binary 0).

Nodes h, b_1 and b_2 are different for each HUB gadget and therefore cannot appear in any external route received by i. This guarantees that the output route cannot share any node (besides d) with the input route, which in turn keeps BGP's loop prevention mechanism from being triggered. More precisely, if i receives no route from its neighboring node p, i selects route $(i\ b_1\ d)$. This allows h to select its preferred path $(h\ i\ b_1\ d)$, which in turn makes o unable to select any valid route to d. Otherwise, if p advertises a route $(p \dots d)$ to i, then i selects $(i\ p \dots d)$. As a consequence, h selects $(h\ b_2\ d)$, and o selects $(o\ h\ b_2\ d)$.

Observe that the output node of the HUB gadget can either announce no route, or it can announce a single route which does not depend on the route received by the input node. For this reason, when connecting the output node o of the HUB gadget to the input node i' of another gadget, i' can receive only one route. Moreover, the same property holds for DISAGREE, BAD-GADGET and NOT gadgets. The OR gadget is a little bit different in that it can output two distinct routes. Since we place a HUB gadget to interconnect any two logic components, the maximum number of paths accepted by the input node of any of the logic components introduced in Sections II-A and II-B is two. Hence, the configuration that encodes a logic circuit can be built in polynomial time in the size of the circuit.

Since a combinational logic circuit can encode any logic formula, the fact that eBGP can be used to construct combinational logic circuits intuitively explains why most problems related to BGP are NP-hard. In fact, by encoding a logic formula in BGP, it is typically possible to obtain a polynomial reduction from SAT [13], a well-known NP-complete problem.

III. UNDERSTANDING THE COMPLEXITY OF BGP USING LOGIC GATES

The fact that eBGP configurations can simulate logic circuits has several implications in terms of the computational complexity of routing problems. Such results depend on how BGP dynamics are modeled. In this section, we separately consider two different models: (i) min-max asynchronous model,

where messages traversing a link have a propagation delay between a minimum and a maximum value; and (ii) purely asynchronous model, where messages can be arbitrarily (even if not indefinitely) delayed. The former is meant to describe the propagation and computing delays in real networks, while the latter is meant to capture every possible message timing, which is useful from the perspective of a protocol designer.

A. Building a Turing Machine with Logic Gates

In the min-max BGP model, each link l is associated with a propagation delay that can take any value within range (m_l, M_l), where m_l (M_l, resp.) is the minimum (maximum, resp.) delay value for l. Both m_l and M_l are finite values. Observe that, if $m_l = M_l$ all BGP message exchanges are completely synchronized. In general, however, we assume $m_l \neq M_l$.

We now prove that in the min-max model a BGP network can simulate a *Finite Turing Machine* (FTM) [14], which is a Turing Machine where the size of the tape is finite. This enables us to show that BGP routing problems are at least as difficult as problems solved by an FTM (i.e., PSPACE-hard). The proof consists of two steps. In the first step we show that it is possible to construct an FTM starting from logic gates and a clock, and interconnecting them with links having a bounded delay. In the second step we use the building blocks in Section II-A to translate the logic circuit that simulates the FTM in an eBGP configuration.

Technically, an FTM is a device that processes symbols on a finite-length tape according to a history-less transition function δ. An FTM maintains a state during the computation and uses a head to read and write on specific cells of the tape. At the beginning, the tape is initialized with an initial string of symbols and the FTM is in an initial state. At each step, the FTM reads the symbol stored in the cell pointed by the head of the tape. Next, according to its state and the read symbol, the transition function δ computes a new symbol to be written on the tape, a new state for the FTM, and eventually a movement of the head of the tape. The computation halts when the FTM reaches some special states.

Now, we build an FTM using only logic gates and a clock (see Fig. 4) and connecting the circuital components with links having a minimum and maximum delay. Each building block in the figure represents a logic circuit, hence it can be encoded with an eBGP configuration (see Section II). Details of the circuits represented in Fig. 4 are provided in Appendix B.

The main issue is the synchronization. Indeed, on one hand, an FTM performs the computation in a centralized way, where all the operations are synchronized by the FTM itself. On the other hand, in a logic circuit the computation is distributed among the different logic gates and only partially synchronized since the signal propagation delay is variable. In a perfectly synchronized model where signal propagation delay is deterministic, it is easy to keep the circuit synchronized by conveniently assigning specific delays to the links. However, in the more general min-max model, the synchronization requires more attention since we need to be sure that the variability of

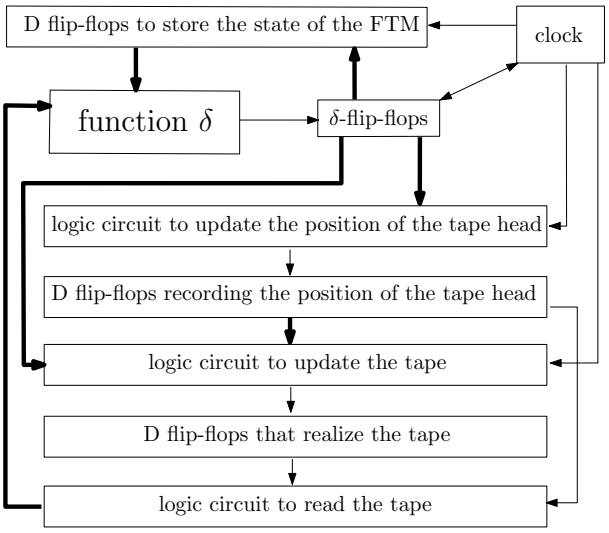


Fig. 4. A Turing Machine. Delays with values $(\frac{2}{3}T, \frac{2}{3}T+\epsilon)$ and $(\epsilon, 2\epsilon)$ are associated with thick and thin lines, respectively, where T is the maximum time the clock holds the same output value.

link delays does not add up across multiple iterations. We build the logic circuit that simulates an FTM as follows. Function δ is simulated by a combinational logic circuit obtained by conveniently interconnecting simple logic gates. We call such a combinational circuit "δ block". A set of D flip-flops, called the δ-flip-flops, are interconnected to the output of the δ block in order to store its output value. The tape, the position of the head of the tape, and the state of the FTM, are simulated using D flip-flops. We build a logic circuit that updates the position of the head and we connect it to the clock (which acts as an enabler), to the δ-flip-flops that store the new position of the head (which is the input of the circuit), and to the D flip-flops that store the current position of the head (which is the output of the circuit). Similarly, we build a circuit that reads the tape and another one that updates it. The link delays within the clock are set in such a way that the clock's output node produces the same output value (either a 0 or a 1) for a time that is at most T and at least $T - \epsilon$, with $\epsilon << T$.

We now show that even if the clock period is variable, we can synchronize the circuit to simulate a FTM. Assume, for the sake of simplicity, that the clock switches to 1 at time 0 and that at time 0 the δ-flip-flops store the new state, the symbol that needs to be written on the tape, and the direction in which to move the head.

Consider a single clock period, which consists of two clock ticks (one for 1 and one for 0). The main intuition is to use the clock value 0 as a disabler, in order to block the propagation of signals until the clock switches to 1 again. During the first tick, the clock outputs 1 so all the δ flip-flops are enabled and signals can propagate. Within this tick, we need to

- propagate the new symbol to write on the tape to the logic circuit that updates the tape;
- propagate the new tape head position to the logic circuit that updates the position;

- propagate the new state to the D flip-flops that store the state; and
- transfer the symbol which is currently on the tape to the input of the δ-block.

Observe that we need to make sure that the new symbol is written on the correct cell of the tape. In order to do so, we introduce a propagation delay between the D flip-flops that store the head position and the logic circuit that updates the tape. This ensures that when the circuit updates the tape, it will refer to the correct position, independent of whether the new position has been already written to the D flip-flops that store the head position.

On the other hand, we need to avoid the possibility that two or more symbols are read and elaborated within a single clock period. In particular, we want the δ-block to be disabled when the newly read symbol arrives, because we want this new symbol to be processed in the next clock period. This means that the propagation delay from the logic circuit that reads the tape to the δ-block must ensure that the symbol arrives when the clock has already switched to 0.

Now, if we can find an assignment of link delays that guarantees the above properties, then we can apply the same arguments to the next clock period and so on, yielding the ability to synchronize the signals to the clock periods and hence completing the definition of FTM. We show an example of such an assignment of link delays in Appendix D.

The computation of an FTM finishes when a *final state* is reached. If and only if the FTM reaches a final state, the clock is stopped by the δ-flip-flops, hence the eBGP configuration stabilizes. Some final states are called *acceptance states*.

Finally, we note that, since an FTM has a tape length which is polynomial in the size of the input string, the entire construction of the logic circuit takes polynomial time. More details about the initialization phase of the logic circuit can be found in [15].

B. Complexity of Min-Max Asynchronous BGP

We now prove that the above logic circuit and the clock can be constructed using an eBGP configuration. The discussion of Section II-A shows that using eBGP we can construct logic gates. Since we are using a BGP model with min-max delays, then we can simply assign the desired delay to BGP peerings. Also, a clock with the appropriate timing can be built interconnecting, as in Section II-A, 3 NOT gates.

The ability to simulate FTMs with eBGP configurations enables us to prove PSPACE-hardness results for BGP problems. We reduce those problems from the LINEAR SPACE ACCEPTANCE problem, which is known to be PSPACE-complete [16]. An instance of LINEAR SPACE ACCEPTANCE consists of a FTM M and a finite string x, where the size of the tape of M is polynomial with respect to the size of x. The problem is to verify if M accepts x. We say that an FTM M accepts a string x if M halts on an acceptance state given that x is initially written on its tape.

In the following, we prove that both SAFETY [2] and REACHABILITY [17] are PSPACE-hard. SAFETY is the problem of deciding whether the BGP network is guaranteed to converge to a stable state.

Theorem 1: SAFETY is PSPACE-hard.

Proof: We reduce SAFETY from the LINEAR SPACE ACCEPTANCE problem. A similar construction with respect to that described above enables to build an eBGP configuration that simulates an arbitrary FTM M in such a way that the network converges to a stable state if and only if M reaches an acceptance state. This polynomial-time reduction directly yields the statement. ■

A very similar reduction from LINEAR SPACE ACCEPTANCE can be leveraged to show the complexity of the REACHABILITY problem [17], that is, deciding whether a BGP configuration admits a stable state in which a given node s has a route to a given destination d. Namely, it is sufficient to build an BGP configuration that simulates the FTM M as shown in the proof of Theorem 1 and modify it such that node s is guaranteed to have a route if and only if the BGP gadget simulating the clock has stopped oscillating.

Theorem 2: REACHABILITY is PSPACE-hard.

The theoretical results above hold also in the case the minimum and maximum delay have the same value (see Appendix D for more details). In this case, there are some interesting practical consequences on BGP simulators. A BGP simulator can be seen as a program that walks through all the possible states of the network. In SPP, a simulator counts how many steps the simulation has performed using a counter. The maximum value of such a counter is set to the number of possible states of the simulated network. In SPP, such a counter has size polynomial w.r.t. the size of the BGP network. When the counter overflows, the simulator has passed through one state at least twice, which means that the network oscillates. Hence, a simulator can be used to solve SAFETY using only a polynomial amount of space, which means that SAFETY is in PSPACE. The fact that SAFETY and REACHABILITY are PSPACE-hard therefore implies that no static analysis algorithm can be (asymptotically) more efficient than running a simulation.

Observe that, theoretically, an infinite BGP network would be able to simulate a Turing Machine with infinite tape. In a sense this means that, despite the simplifications listed in Section II, unrestricted BGP policies have the same expressive power as Turing Machines.

C. Complexity of Pure Asynchronous BGP

By assuming complete asynchronism, messages are allowed to be delayed arbitrarily, even if not indefinitely. This means that every message must be eventually delivered, but no constraint is imposed on when the delivery occurs. Observe that the pure asynchronous model is not a special case of the min-max model, because the upper bound on each link cannot be assigned to a finite value as messages can be arbitrarily delayed.

Our results on the mapping between BGP and logic gates allow to simply prove complexity results on BGP even in this pure asynchronous setting. For instance, consider the

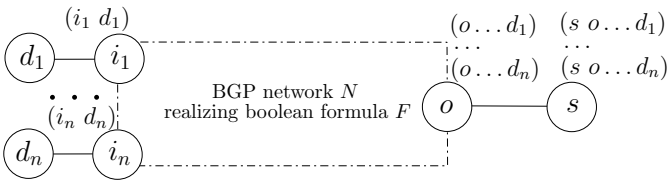


Fig. 5. Scheme of the reduction from SAT COMPLEMENT to MOAS REACHABILITY.

following problem, called MOAS REACHABILITY. Assume that a destination prefix is generated by multiple origin ASes in the eBGP network, as it happens when IP anycast is deployed in the Internet (e.g., for the DNS root name server). The MOAS REACHABILITY problem consists in determining if the destination prefix is reachable from a given source AS for any nonempty subset of origin ASes announcing the prefix. Such a problem aims at verifying that reachability of a given MOAS (Multiple Origin AS) destination is guaranteed in presence of failures or planned maintenance.

Leveraging the mapping between BGP networks and logic gates, it is easy to show that MOAS REACHABILITY is NP-hard. Note that we cannot prove that the problem is PSPACE-hard since the model makes it impossible to simulate a clock with a given period and enables us only to build combinational logic circuits. The scheme of the reduction from SAT COMPLEMENT [13] is represented in Fig. 5, where nodes labeled as d_i, with $i = 1, \dots, n$, represent the origin ASes announcing the destination prefix, and N, F, and s are defined as follows.

Let F be the boolean formula in conjunctive normal form that represents an instance of SAT. Let s be a vertex that is the given source AS. We interpose between them a BGP network N which simulates the logic circuit corresponding to F.

The reduction can be built in polynomial time with respect to the size of F. Indeed, given that the number of clauses in F is C, the number of origin ASes d_i is equal to C, and each gate in N has a constant number of nodes, each accepting at most $2 * C$ paths, because of the presence of HUB gadgets at each interconnection between gates. Now, considering the combination of origin ASes from which the prefix is announced corresponds to providing all possible inputs to the network N. By construction of N, this translates to considering all boolean assignments to variables in the original boolean formula F. Hence, s receives a route for each combination of origin ASes announcing the destination prefix if and only if the boolean formula F is satisfied by any boolean assignment. The following theorem follows from the coNP-hardness of SAT COMPLEMENT.

Theorem 3: MOAS REACHABILITY is coNP-hard.

We stress that, by applying the same reduction technique, it is also straightforward to build reductions for problems like REACHABILITY, ASYMMETRY, SOLVABILITY, TRAPPED, UNIQUE, and MULTIPLE [17]. Details are provided in Appendix A.

IV. THE IMPACT OF POLICY RESTRICTIONS

Intuitively, the fact that BGP configurations can encode arbitrary logic circuits suggests that the complexity of BGP related problems stem out of the intrinsic complexity of BGP semantics, which ultimately maps to the expressiveness of BGP policies. One might argue that it is not surprising that completely unrestricted policies yield complex semantics. It is therefore interesting to study whether restricting BGP policies significantly simplifies the analysis of a BGP configuration.

In this section, we consider iBGP configurations, where policies are dictated by the iBGP route propagation rules and IGP distances, Local Transit policies, where policies depend solely on the ingress and egress AS, and Gao-Rexford conditions, where policies are tied to commercial relationships among ASes.

A. Restricting to iBGP

As opposed to eBGP, in iBGP all routers belong to the same AS. For this reason, routing policies are typically not applied on iBGP messages [3]. However, the specification of iBGP with route reflection imposes an implicit route ranking and an implicit route filtering. The ranking component is restricted in that it has to be consistent with the IGP graph. The filtering component is restricted in that it has to be consistent with the iBGP graph.

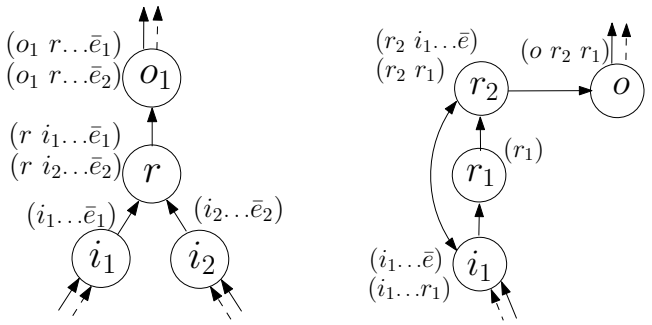
In particular, the BGP decision process has some tie breaking rules that only have local significance (e.g., IGP distance), therefore routing preferences are implicitly imposed by the BGP decision process itself. Further, when route reflection [18] is used, the protocol requires certain routes to be filtered out at each iBGP router. More precisely, the iBGP neighbors of each router are split into three sets: *clients*, *peers* and *route-reflectors*. Best routes are always relayed to clients, but best routes learned from peers or route-reflectors are not propagated to other peers and route-reflectors.

We now show that, despite the restrictions above, the protocol still retains enough expressive power to encode arbitrary logic circuits.

First of all, observe that we can consider just egress points preferences, disregarding the details of the IGP graph. In fact, it can be shown that for any given set of egress point preferences there exists an IGP graph which is consistent with those preferences. The algorithm to build such an IGP graph is described in Appendix E.

Fig. 6 depicts iBGP configurations that correspond to OR and NOT gates. One-headed solid arrows represent sessions from a client to its route reflector, while double-headed solid arrows represent sessions between two peers. Inbound and outbound dashed arrows indicate input and output nodes, respectively. Paths aside each router represent the iBGP path towards egress points. The rest of the notation is consistent with the graphical convention introduced in Section II.

The iBGP configuration in Fig. 6(a) simulates the behavior of an OR logic gate. The output router will receive a route to any of the egress points $\bar{e}_1$ and $\bar{e}_2$ if and only if a route is received by either i_1 or i_2 (or both). Similarly, Fig. 6(b) depicts



(a) The OR gate in iBGP. (b) The NOT gate in iBGP.

Fig. 6. iBGP configurations which simulate logic gates in iBGP.

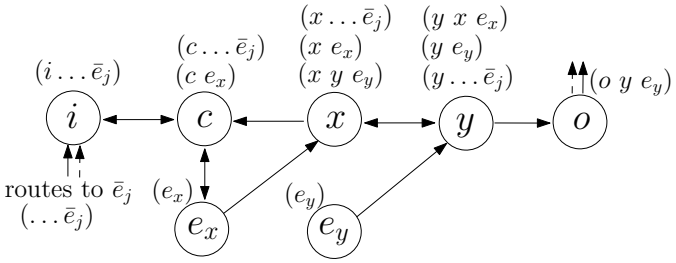


Fig. 7. The iBGP-HUB gadget.

an iBGP configuration corresponding to the NOT gate. If i_1 receives an eBGP route, it will propagate it to r_2. Because of egress point preferences, r_2 will select the route announced by i_1. Now, since this route was learned from a peer, iBGP route propagation rules require that r_2 do not relay the route to o, which therefore is unable to learn any feasible route. On the contrary, if i_1 receives no route, r_2 will select the route announced by r_1 and will propagate it to o. Note that the iBGP configuration corresponding to the NOT gate is based on the OVER-RIDE gadget introduced in [19].

Reference [3] shows examples of iBGP configurations realizing DISAGREE and BAD-GADGET structures. This enables us to build the memory and the clock components as we did for eBGP in Section II.

Finally, to interconnect the logic components, we need the equivalent of the HUB gadget for iBGP. Consider the iBGP-HUB gadget depicted in Fig. 7. If i receives an iBGP path R towards any egress point $\bar{e}_j$, then i, c and x also select route R. In this case, router y selects path $(y\ e_y)$ because of its egress point preferences, and propagates it to o. Hence, o receives, selects and propagates one route which has no router in common with the original route R. Otherwise, if i receives no path towards any $\bar{e}_j$, then x selects route $(x\ e_x)$, enabling y to select its most preferred route $(y\ x\ e_x)$. However, y cannot propagate its best path to o because of iBGP propagation rules that deny propagation of a path learned from an iBGP peer to a route reflector.

Observe that the iBGP-HUB gadget outputs at most one route, and has at most two routes in input. Also, node i cannot receive paths from c, which implies that routes can only flow from the left to the right part of the gadget only, hence preventing propagation of routes in undesired directions. In fact, either i) node i selects a path $R = (i \ldots \bar{e}_j)$ which is learned over the client session itself; or ii) node i has no path to select as c's best route $(c\ e_x)$ is learned from an iBGP peer and cannot be propagated to another iBGP peer.

As a consequence, we can derive the computational intractability results (NP-hardness for the asynchronous model and PSPACE-hardness for the min-max model) of all correctness problems defined in iBGP, namely signaling, dissemination and forwarding correctness [3], [19]. Having all the needed building blocks, the iBGP configuration that simulates a Finite Turing Machine can be done exactly as in Section III for eBGP configurations. Regarding the NP-hardness proofs in the asynchronous model, the reduction method described in Section III can be re-used, with the only difference that the set of possible boolean assignments to variables in the SAT formula should be mapped into the set of egress points receiving an eBGP route to the considered prefix.

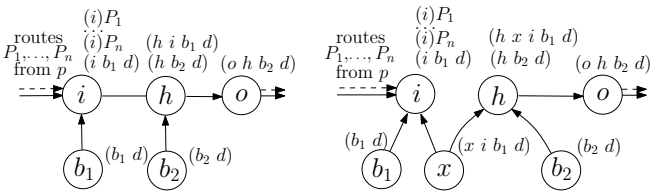
B. Local Transit Policies and Gao-Rexford Conditions

We now consider eBGP policy restrictions that have been proposed in the literature.

A common policy configuration practice consists in applying the so-called Local Transit policies [11]. Local Transit policies consist in defining routing policies as functions of the AS that announces the route and of the AS to which the route is announced only. Observe that all the policies used to build the gadgets presented in Section II are compliant with the definition of Local Transit policies. The same holds for the DISAGREE gadget and the BAD-GADGET. As a consequence, BGP problems remain hard (in the asynchronous model) or very hard (in the min-max model) even for BGP networks in which only Local Transit policies are applied. These results extend the findings in [12].

A further restriction with respect to Local Transit Policies consists in imposing that the eBGP configuration satisfies the Gao-Rexford conditions introduced in [6]. These conditions are the most famous way to trade policy expressiveness for correctness guarantees without the need for global coordination among ASes. Gao-Rexford conditions assume that each AS classifies its eBGP neighbors as either customers, peers, or providers, and that: i) routes learned from customers are preferred over those learned from peers and providers; ii) there is no cycle such that each AS in the cycle is a customer of the next AS in the cycle; iii) an AS does not export routes learned from a peer or provider to its peers or providers. It has been proved [6] that the Gao-Rexford conditions guarantee that BGP always converges to a unique stable state. Thus, a greedy algorithm like the one proposed in [2] can be used to compute the stable state, and to solve BGP problems in polynomial time.

The BGP networks simulating the NOT and OR logic gates (Fig. 1) are compliant with Gao-Rexford conditions if the output node o_1 is set as provider of r, and r is a provider of all the other nodes. Similarly, the HUB gadget (Fig. 3) can be forced to be compliant with the Gao-Rexford conditions if



(a) A HUB violating Condition i). (b) A HUB violating Condition iii).

Fig. 8. Variants of the HUB gadget obtained by violating one of the Gao-Rexford conditions. An oriented (unoriented) edge from a to b represents the fact that a is a customer (peer) of b.

o is a provider of h, h is a provider of both i and b_2, and i is a provider of both p and b_1. This assignment of commercial relationships has the property that if a logic circuit does not contain cycles (as in combinational circuits) then it can be simulated by a BGP network that satisfies the Gao-Rexford conditions. Otherwise, cycles in the logic circuits translates to customer-provider cycles in the eBGP configuration, which violates the second Gao-Rexford condition. The argument above implies that the MOAS REACHABILITY problem introduced in Section III remains NP-hard even when Gao-Rexford conditions are enforced. This can be shown by simply re-using the same proof as in Section III. It is interesting to note that MOAS REACHABILITY is NP-hard under Gao-Rexford conditions, because other BGP problems are polynomial in such a setting. However, assuming Gao-Rexford conditions prevents us from building arbitrary logic circuits and configurations like a DISAGREE or a BAD-GADGET. This can be seen as an intuitive explanation of why most BGP problems turn out to be polynomial in such a setting.

However, violating any of the Gao-Rexford conditions enables us to build configurations that simulates arbitrary logic circuits, hence arbitrary Finite Turing Machines in the min-max model. We have already shown a customer-provider assignment such that a cycle in a logic circuit translates to a customer-provider cycle in the BGP network. Hence, if we violate condition ii) and customer-provider loops are allowed, then every interconnection between logic components is admitted. Otherwise, if conditions i) or iii) are violated, we modify the HUB gadget as shown in Fig. 8. In each of these two cases, cycles in the logic circuits are guaranteed not to translate to customer-provider cycles, and only one of the Gao-Rexford conditions is violated at the time. Hence, for any violation of the Gao-Rexford conditions, arbitrary logic circuit can be simulated with BGP configurations. As a consequence, convergence and route propagation problems are still PSPACE-hard if any of the Gao-Rexford condition is violated.

An interesting problem that remains open is whether policy restrictions exist that do not guarantee convergence but allow efficient analysis of BGP configurations.

V. RELATED WORK

Previous work has mostly focused on the asynchronous BGP model in which messages can be arbitrarily (even if not indefinitely) delayed. Several BGP routing problems have been shown to be computationally intractable for both eBGP and iBGP, in different models for expressing routing policies [17], [2], [3].

Recent work focused on determining the computational complexity of the fundamental SAFETY problem. We recall that SAFETY consists in determining whether a BGP network is guaranteed to converge. In [20] the problem is claimed to be polynomial time solvable if "spurious" updates are admitted. In [21] SAFETY is proved to be PSPACE-complete in an unrealistic game-theoretical model in which BGP speakers are assumed to be omniscient and BGP messages are not passed router by router. In [12] the impact of models forcing different kinds of policy restrictions on the computational complexity of SAFETY and related problems has been studied. Our work shows that SAFETY is PSPACE-hard in a model in which bounded link delays are associated to each link. Further, Section IV contains a study similar to [12] in which, however, we consider different BGP problems and more policy restrictions.

In [22] BGP is modeled as a specific game where each router is a player and the router preferences are mapped to the players's strategies. The authors proved that for general players's strategies, the problem of determining if the dynamics of the game halts is PSPACE-complete in a pure asynchronous setting. However, this results does not extend where players's strategies are more specific, as in BGP.

VI. CONCLUSIONS

Over the last 15 years, a routing theory has been developed to study problems on BGP convergence and route propagation. In this paper, we extend the current theory by describing a mapping between BGP configurations and logic circuits. Also, we show how to leverage the mapping to devise reduction techniques and define the computational complexity of several BGP routing problems in different models. Most notably, by simulating Finite Turing Machines with BGP configurations, we prove the PSPACE-hardness of famous BGP problems like REACHABILITY and SAFETY in a model in which link delays are bounded into ranges of values.

We finally investigate the impact of BGP policy restrictions on the possibility to build our mapping. Whenever convergence is not guaranteed, the protocol resulted to be still powerful enough for BGP configurations to simulate arbitrary logic circuits, implying that BGP problems remain computationally intractable in the considered models.

In future work, we plan to investigate whether policy restrictions exist that do not guarantee convergence but allow efficient analysis of BGP configurations. We also plan to extend our analysis to other models and routing protocols.

REFERENCES

[1] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, 2006.
[2] T. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Trans. on Networking*, 2002.
[3] T. Griffin and G. T. Wilfong, "On the correctness of ibgp configuration," in *Proc. SIGCOMM*, 2002.

[4] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP misconfiguration," in *Proc. SIGCOMM*, 2002.

[5] N. Kushman, S. Kandula, and D. Katabi, "Can you hear me now?! It must be BGP," in *Computer Communication Review*, 2007.

[6] L. Gao and J. Rexford, "Stable internet routing without global coordination," in *Proc. SIGMETRICS*, 2000.

[7] A. Flavel and M. Roughan, "Stable and Flexible iBGP," in *Proc. SIGCOMM*, 2009.

[8] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with c-bgp," *IEEE Network*, vol. 19, no. 6, 2005.

[9] A. Flavel, J. McMahon, A. Shaikh, M. Roughan, and N. Bean, "BGP Route Prediction within ISPs," *Comput. Commun.*, vol. 33, 2010.

[10] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *STTT*, vol. 7, no. 2, pp. 156–173, 2005.

[11] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica, "Pathlet routing," in *Proc. SIGCOMM*, 2009.

[12] M. Chiesa, L. Cittadini, G. Di Battista, and S. Vissicchio, "Local Transit Policies and the Complexity of BGP Stability Testing," in *Proc. INFOCOM*, 2011.

[13] C. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994.

[14] P. Linz, *An Introduction to Formal Language and Automata*. USA: Jones and Bartlett Publishers, Inc., 2006.

[15] M. Chiesa, L. Cittadini, G. Di Battista, L. Vanbever, and S. Vissicchio, "Computing with BGP: from Routing Configurations to Turing Machines," Université Catholique de Louvain, Tech. Rep., 2012, http://dial.academielouvain.be/handle/boreal:113003?site_name=UCL.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[17] T. G. Griffin and G. Wilfong, "An Analysis of BGP Convergence Properties," in *Proc. SIGCOMM*, 1999.

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

[19] S. Vissicchio, L. Cittadini, L. Vanbever, and O. Bonaventure, "iBGP Deceptions: More Sessions, Fewer Routes," in *Proc. INFOCOM*, 2012.

[20] M. Suchara, A. Fabrikant, and J. Rexford, "Bgp safety with spurious updates," in *Proc. INFOCOM*, 2011, pp. 2966–2974.

[21] A. Fabrikant and C. Papadimitriou, "The complexity of game dynamics: Bgp oscillations, sink equilibria, and beyond," in *Proc. SODA*, 2008.

[22] A. D. Jaggard, M. Schapira, and R. N. Wright, "Distributed computing with rules of thumb," in *PODC*, 2011, pp. 333–334.

APPENDIX

A. Reductions for the Asynchronous BGP Model

In the following, we show how to leverage the mapping between eBGP configurations and logic gates to prove the complexity of the problems studied in [17]. All the following reductions can be built in polynomial time with respect to F, since the eBGP configuration corresponding to each gate has a constant number of nodes, and each node accepts a number of paths bounded to two (see Section II-C). Also note that a DISAGREE can be obtained as a loop of two NOT gadgets. As a consequence, the following complexity proofs remain valid whenever the OR and the NOT logic gates can be simulated and arbitrarily interconnected via the HUB gadget.

1) Reachability: The REACHABILITY problem consists in deciding if a BGP network admits a stable state in which a given AS s has a route to a given destination AS d. The problem has been already shown in [17] to be NP-hard. We now show a simpler NP-hardness proof of the problem based on the mapping between BGP configurations and logic gates.

Theorem 4: REACHABILITY is NP-hard.

Proof: Let F be the boolean formula in conjunctive normal form that represents an instance of SAT. We build the instance of REACHABILITY as follows. Refer to Fig. 9. Let s and d be the source and the destination ASes considered

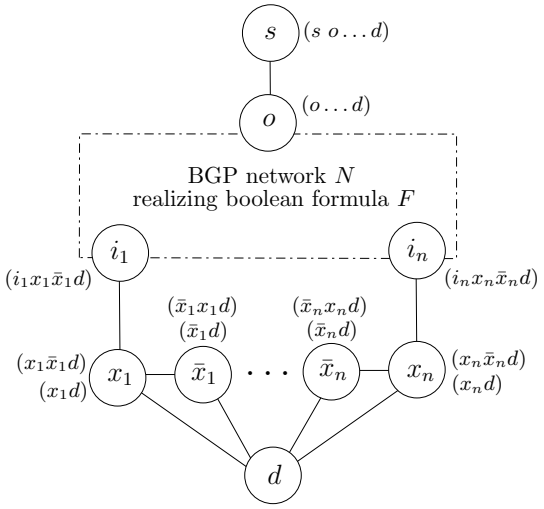


Fig. 9. Scheme of the reduction from SAT to REACHABILITY.

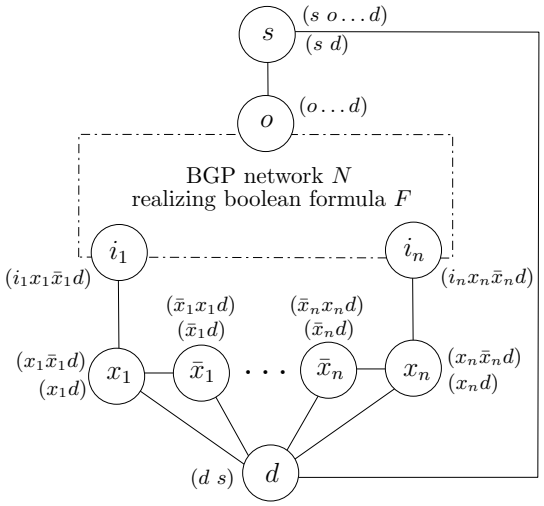


Fig. 10. Scheme of the reduction from SAT to ASYMMETRY.

in REACHABILITY, respectively. We interpose between them a BGP network N which simulates the logic circuit corresponding to F. We also add a DISAGREE gadget between each input router i_j in N and the destination d. Since each of the n DISAGREE gadgets can independently converge to one of two distinct stable states, we have 2^n possible combinations of stable states, each of which is mapped to a boolean assignment of the variables. For each possible stable state, by construction of the BGP network N, node o will have a route to d if and only if formula F is satisfied by the assignment corresponding to the stable state. The same argument applies to node s, which completes the proof. ■

2) Asymmetry: The ASYMMETRY problem is defined as follows: given a BGP configuration and two ASes s and d, does there exist a stable state in which the path from s to d is not the reverse of the path from d to s? Fig. 10 depicts the scheme of the reduction from SAT. As in the reduction to REACHABILITY, a series of DISAGREEs is interposed between

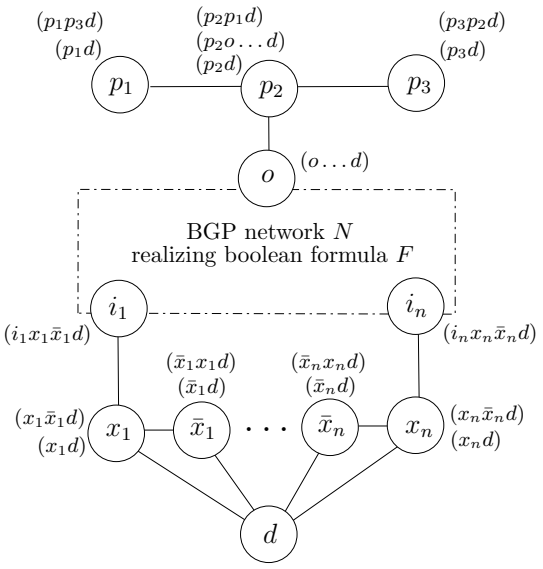


Fig. 11. Scheme of the reduction from SAT to SOLVABILITY.

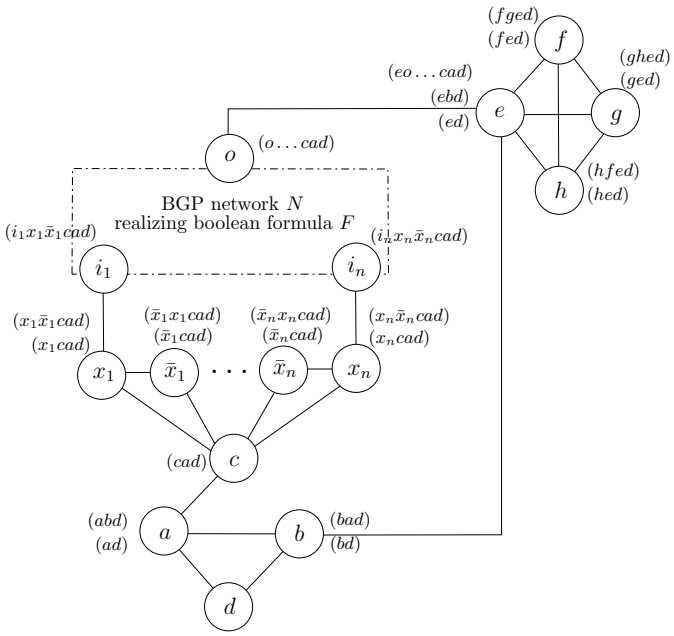


Fig. 12. Scheme of the reduction from SAT to UNIQUE.

node d and a BGP configuration realizing the input formula F. Path selection at node s depends on the output of the network N.

Theorem 5: ASYMMETRY is NP-hard.

Proof: Consider the reduction described above. Let s and d be the pair of ASes considered in ASYMMETRY problem. Node d is guaranteed to select $(d\ s)$, since it is the only path that d accepts to s and the path is always available at d (because of the direct link between s and d).

Now consider the path s selects to reach d. Since each of the n DISAGREE gadgets can independently converge to two distinct stable states, we have 2^n possible combinations of stable states, each of which is mapped to a boolean assignment of the variables in F. For each possible stable state, by construction of the BGP network N, node o will have a route to d if and only if formula F is satisfied by the assignment corresponding to the stable state. Hence, s can choose its preferred path $(s\ o\dots d)$ if and only if F is satisfiable. Otherwise, s will have to backup on $(s\ d)$.

To summarize, if F is satisfiable then there exists a state in which s does not select $(s\ d)$, which is the reverse of the path from d to s. The NP-completeness of the SAT problem then yields to the statement. ∎

3) Solvability and Trapped: We now consider the SOLVABILITY and TRAPPED problems that deal with convergence guarantees of a BGP configuration. Namely, SOLVABILITY consists of deciding if a given BGP configuration admits at least one stable solution, while TRAPPED is the problem of deciding if the network can be trapped in permanent routing oscillations, like those occurring in a BAD-GADGET. For both problems, we use the reduction represented in Fig. 11. In the reduced instance, a BAD-GADGET exists between nodes p_1, p_2, and p_3. The BAD-GADGET is prevented from oscillating if and only if p_2 receives path $(o\dots d)$ from o. However, by construction of the reduced instance, deciding if o selects path $(o\dots d)$ corresponds to decide if a boolean assignment that satisfies F exists. As a result, the following two theorems hold.

Theorem 6: SOLVABILITY is NP-hard.

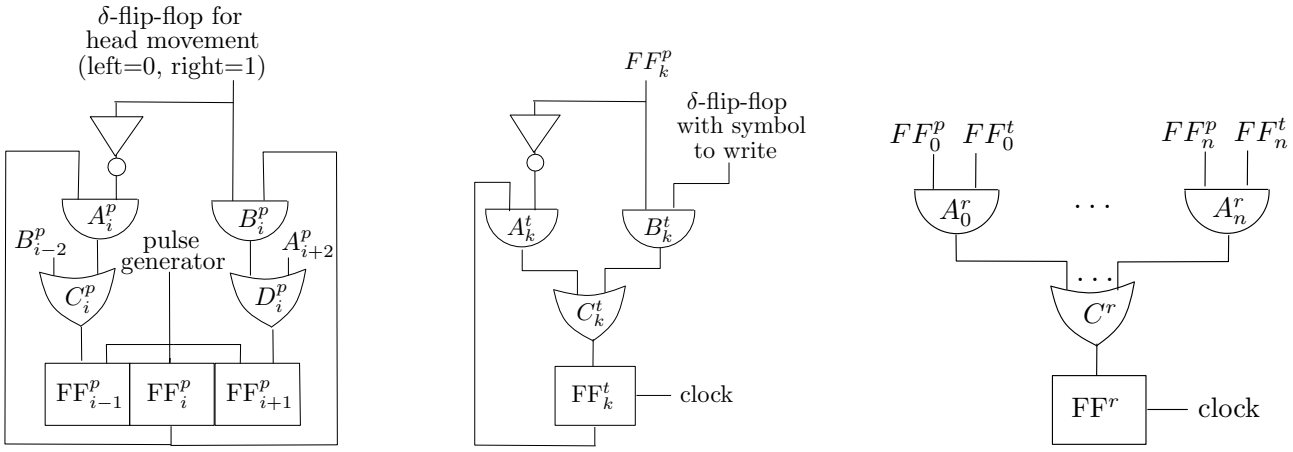
Theorem 7: TRAPPED is NP-hard.

4) Unique and Multiple: UNIQUE is the problem of deciding if a single stable state exists for a given BGP configuration (as opposed to the existence of multiple stable states). Fig. 12 shows how to prove the NP-hardness of the UNIQUE problem, applying again our reduction technique. In the reduced instance in Fig. 12 the presence of a stable state is guaranteed in one of the two stable states of the DISAGREE between a and b. Indeed, if a steadily selects $(a\ b\ d)$ and b steadily selects $(b\ d)$, then no route is provided to c, which implies o having no route, and e selecting $(e\ b\ d)$. This, in turn, implies nodes f, g, and h having no route to d. However, if a steadily selects $(a\ d)$ and b steadily selects $(b\ a\ d)$, then the presence of additional stable states depends on the formula F. Indeed, if F is not satisfiable then o can never have a route. In this case, e is forced to select $(e\ d)$ activating the BAD-GADGET between f, g, and h. As a consequence no other stable state exists. Otherwise, if F is satisfiable then o can steadily select a route for at least one combination of inputs to N. This involves that e will select the route it receives from e, hence preventing the BAD-GADGET between f, g, and h from oscillating, and forcing a second stable state. This allows us to formulate the following theorem.

Theorem 8: UNIQUE is NP-hard.

The NP-hardness of UNIQUE directly implies the NP-hardness of MULTIPLE, as already noted in [17].

Theorem 9: MULTIPLE is NP-hard.



(a) Logic circuit to update the position of the tape head.

(b) Logic circuit to update the tape.

(c) Logic circuit to read the tape.

Fig. 13. Building Blocks of the Turing Machines.

B. Details on the Turing Machine Building Blocks

In this appendix, we provide more details on the eBGP configuration corresponding to a given Finite Turing Machine (FTM). In particular, we illustrate the logic circuits in Fig. 4, except the one realizing the δ function which strongly depends on the given FTM to be simulated.

The logic circuit to update the position of the head is depicted in Fig. 13(a), where FF_i^p represents a D flip-flop which outputs a 1 if the head is currently on the i-th cell of the tape. Before turning on the Turing machine, all flip-flops FF_i^p store a 0 except FF_0^p, because the head is on the first cell of the tape. The logic circuit in Fig. 13(a) ensures that, throughout the computation, only one flip-flop FF_i^p can store a 1 at any given time. Intuitively, this models the fact that the head can only be in a single position at any given time. The flip-flops are enabled by a simple circuit that generates a pulse (i.e., a 1 immediately followed by a 0) each time the clock has a transition from 0 to 1. The input of the circuit is the output signal of the δ-flip-flop FF' which stores the next movement of the tape head. Without loss of generality, we assume that the transition function δ of the Turing machine has no transitions where the tape head remains on the same cell.

Since the AND gates A_i^p and B_i^p take FF_i^p as input, only one AND gate outputs a 1: for example, if the tape is on cell k and the output of the δ function is 0, only A_k^p will output a 1, while all other AND gadgets will output a 0. The OR gates allow us to write a 1 in FF_j^p i) when the head is at position j_1 and it must move to the right; or ii) when the head is at position $j+1$ and it must move to the left.

We now show how the logic circuit in Fig. 13(a) ensures that if FF' stores a 0 (1, resp.) then the head must move to the cell which is at the left (right, resp.) of the current cell. Indeed, consider the case in which the current position of the head is k, and the output of FF' is 0. In this case, only FF_k^p stores a 1, until time $\bar{t}$ at which the clock makes a transition from 0 to 1. At time $\bar{t}$, all the AND gates A_i^p and B_i^p such that $i \neq k$ output a 0, which implies $FF_i^p = 0$ for all $i < k-1$ and $i > k+1$. Moreover, $A_{k+2}^p = 0$ since $FF_{k+2} = 0$, and B_k^p since $FF' = 0$, which imples $FF_{k+1}^p = 0$. We have $A_k = 1$ which implies $FF_{i-1}^p = 1$. A symmetric argument can be applied to show that $FF_{i+1}^p = 1$ if $FF' = 1$.

Fig. 13(b) shows the logic circuit used to update the tape. The input of this circuit is represented by the symbol to write on the tape (as stored in one of the δ-flip-flops) and the output of the flip-flops representing the position of the head on the tape. The figure illustrates only the part of the circuit related to one cell of the tape, represented by the flip-flop FF_k^t. Consider now the dynamic behavior of the circuit. When FF_t^k is disabled by the clock (clock at 0), it outputs its stored value. When it is enabled by the clock (clock at 1), the following cases can occur.

- FF_k^p outputs 0, meaning that the current position of the head is j with $j \neq k$. In this case, FF_k^t continue to store the same value as before. Indeed, the output of B_k is 0 no matter what is the symbol to write. Hence, the output of C_k depends only on the output of A_k, and since $FF_k^p = 0$, $A_k = FF_k^t$.
- FF_k^p outputs 1, meaning that the current position of the head is k. In this case, the new symbol is written on FF_k^t. Indeed, the output of A_k is 0 since one of this input (NOT FF_k^p) is 0. Thus, the output of C_k depends only on the output of B_k, which, in turn, coincides with the symbol to write.

Finally, Fig. 13(c) illustrates the logic circuit which provides the symbol stored in the current cell of the tape as input to the circuit realizing the δ function. Such a symbol is stored in a flip-flop FF^r which is enabled by the clock. The circuit is based on a set of AND gates A_i^r, and a single OR gate C^r. Each AND gate A_i^r receives as input the value stored in FF_i^p and FF_i^t. Assume that the position of the head is k. By definition of the flip-flops FF_i^p that implement the head

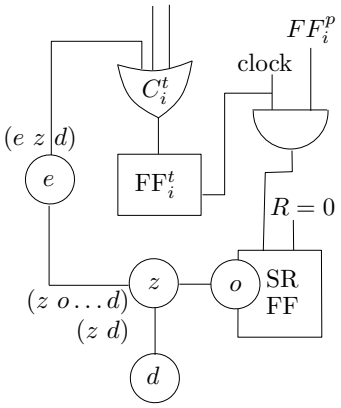


Fig. 14. Initialization of the flip-flops representing the tape.

position, FF_k^p stores a 1, while all the others store a 0. Hence, $A_i = 0$ for all $i \neq k$. A_k will output the same value stored by FF_k^t. As a consequence, the value stored in FF^r will be the same as the value of the current cell of the tape FF_k^t.

C. Initialization of the Finite Turing Machine

Some of the BGP problems are defined on precise routing states, e.g., states actually reachable by BGP. Given an FTM and an input string written on its tape, we now show how to build a BGP configuration such that the flip-flops representing the tape are correctly initialized by BGP dynamics before simulating the FTM computation.

Writing the initial value in the tape flip-flops is straightforward: it suffices to have the destination d directly connected to all the flip-flops representing cells of the tape that store a 1. The main problem, however, is to ensure that the δ function is able to overwrite the initial value when it needs to. Since in BGP any route is preferred to the absence of a route, we must carefully handle the case when the δ function must write a 0 (i.e., absence of a route) in a cell that has been initialized with a 1 (i.e., presence of a route).

Fig. 14 depicts how to solve the problem. Consider the OR gate C_i^t. Route $(e\ z\ d)$ is provided by node e as input to FF_i^t. However, z chooses route $(z\ d)$ only if the output of the SR flip-flop $SR - FF$ is 0. Observe that the input signal R of the SR flip-flop $SR - FF$ is always set to 0. This implies that, after the first time that that the S input of the flip-flop is set to 1, the SR flip-flop will always store a 1. Since the S input of $SR - FF$ depends on FF_i^t and the clock, we have that $SR - FF$ will always store a route after cell FF_i^t has been read for the first time. This ensures that the initial value is deleted after the first time the cell is read. The circuit to update the tape will then overwrite the initial value with the output of the δ function.

D. Analysis of the Simulation of a Finite Turing Machine

In Section III-A, we claimed that, in the min-max model, delay can be assigned to links in such a way that each step of a Finite Turing Machine (FTM) can be simulated by the corresponding BGP configuration in a separate clock period. We now prove our claim by showing a convenient link delay assignment.

Delays between the logic circuit blocks are shown in Fig. 4. In the figure, thick and thin lines represent delays of $(\frac{2}{3}T, \frac{2}{3}T + \epsilon)$ and $(\epsilon, 2\epsilon)$ respectively, where $2T$ is the period of the clock and $\epsilon << T$. All the internal delays of the various logic circuit blocks are set to $(\epsilon, 2\epsilon)$. Observe that, since $\epsilon << T$, the ordering of events occurring during one clock period, is not influenced by ϵ link delays or their sum. Hence, we disregard of ϵ delays in the following.

We now show that the ordering of events occurring during one clock period is as defined in Section III-A, allowing us to simulate any FTM in the min-max model. The computation starts at time $t = 0$, with the output of the clock at 0 until $t = T$. Before $t = T$, the only event that occurs is the transfer of the symbol in the current cell of the tape to function δ. At time $t = T$ the clock signal changes from 0 to 1, enabling the δ-flip-flops. As a consequence, the δ-flip-flops change their output values according to the output of the δ function. After $\frac{2}{3}T$, the new output of the δ-flip-flops is provided as input to the logic circuit that controls the tape head, to the logic circuit to update the tape, and to the flip-flops that store the state of the FTM. Hence, at time $t = T + \frac{2}{3}T = \frac{5}{3}T$, the state and the position of the head are updated. Moreover, at time $T + 2\frac{2}{3}T = \frac{7}{3}T$, the tape is updated, and the new symbol on the tape and the new state of the FTM are provided as input to the δ function. Thus, the new output of function δ is ready at time $\frac{7}{3}T$. However, since $\frac{7}{3}T > 2T$ and $\frac{7}{3}T < 3T$, the clock is at 0 and all flip-flops are disabled at that time, and nothing changes anymore until $t = 3T$, when the simulation of the next step of the FTM is performed.

Observe that if $\epsilon = 0$ the arguments above are still valid. This means that the above simulation works also when the minimum and maximum delay on each link are set to the same value, i.e., in the completely synchronous model.

E. The IGP Builder (IB) Algorithm

Consider a single destination d. Let $\mathcal{E}$ be the set of egress points to d. Let $\lambda^v : \mathcal{E} \to \mathbb{N}$ be the egress point ranking function of router v, such that $\lambda^v(e_j) = i$ if and only if e_j is the i-th most preferred egress point by v. Since the BGP process forces each router to deterministically select only one route, egress point preferences at each router are totally ordered, that is, $\forall e_i \neq e_j\ \lambda^v(e_i) \neq \lambda^v(e_j)$. Given the ranking functions of all the routers in the networks, the IB builds the IGP graph as follows. For each pair of a router r and an egress point e, we add a link (r, e) in the IGP graph. To each link (r, e), we assign a weight $w(r, e) = \lambda^r(e) + |\mathcal{E}|$.

We now prove that the IB algorithm is correct, that is, it builds an IGP graph consistent with the given egress point preferences at each router. Let $dist(v, u)$ be the length of the shortest path from v to u. We say that an IGP topology *realizes* the given ranking functions if for each router $v \notin \mathcal{E}$ and each arbitrary pair of distinct egress points e_1 and e_2, $\lambda^v(e_1) < \lambda^v(e_2)$ implies that $dist(v, e_1) < dist(v, e_2)$.

Lemma 1: In the IGP topology built by the IB algorithm, the shortest path between any router r and any egress point e is $(r\ e)$.

Proof: Let $G = (V, E)$ be the IGP topology built by the IB algorithm. Consider any router r and any egress point e. By construction, the weight of the path $(r\ e)$ is equal to $w(r, e) = \lambda^r(e) + |\mathcal{E}| \leq 2|\mathcal{E}|$.

We now show that any path P from r to e, with $P \neq (r\ e)$, has a weight higher than $w(r, e)$. By definition of P, P contains at least two edges. By definition of the weight function adopted in the IB algorithm, the weight of P is equal or greater to $2+2|\mathcal{E}|$. Hence, the weight of any path $P \neq (r\ e)$ is higher with respect to $(r\ e)$, yielding the statement. ■

Theorem 10: Given a set Λ of ranking functions, the IB algorithm builds an IGP topology that realizes Λ.

Proof: Let $G = (V, E)$ be the IGP topology built by the IB algorithm. Consider an router r and any pair of egress points e_1 and e_2, such that r prefers routes from e_1 to routes from e_2. By Lemma 1, $dist(r, e_1) = w(r, e_1)$ and $dist(r, e_2) = w(r, e_2)$. By definition of the weight function adopted in the IB algorithm, we then have $w(r, e_1) < w(r, e_2)$, which proves the statement. ■