applied sciences

MDPI

Article

Policy-Compliant Maximum Network Flows

Pieter Audenaert *, Didier Colle and Mario Pickavet

Faculty of Engineering and Architecture, Department of Information Technology, Ghent-University-imec, 9052 Ghent, Belgium; didier.colle@ugent.be (D.C.); mario.pickavet@ugent.be (M.P.)

* Correspondence: Pieter.Audenaert@UGent.be; Tel.: +32-(0)9-33-14900

Received: 19 January 2019; Accepted: 23 February 2019; Published: 28 February 2019

Abstract: Computer network administrators are often interested in the maximal bandwidth that can be achieved between two nodes in the network, or how many edges can fail before the network gets disconnected. Classic maximum flow algorithms that solve these problems are well-known. However, in practice, network policies are in effect, severely restricting the flow that can actually be set up. These policies are put into place to conform to service level agreements and optimize network throughput, and can have a large impact on the actual routing of the flows. In this work, we model the problem and define a series of progressively more complex conditions and algorithms that calculate increasingly tighter bounds on the policy-compliant maximum flow using regular expressions and finite state automata. To the best of our knowledge, this is the first time that specific conditions are deduced, which characterize how to calculate policy-compliant maximum flows using classic algorithms on an unmodified network.

Keywords: communication networks; maximum flow; network policies; algorithms

1. Introduction

Connecting two nodes in a computer communication network involves setting up paths, possibly more than one. Often, this is done with resiliency in mind: clearly, having more than one path between nodes, it might be possible to route around a failed link or node, depending on the paths themselves and the failed node or link. Also, network operators provide multiple paths between nodes in order to increase the maximum achievable throughput or flow in between these nodes.

In practice, however, network policies are in effect, effectively restricting the flow that can be set up between two nodes in the network. Specific restrictions can be implemented to optimize routing, due to security constraints or because of policies and agreements between different network operators. As one example, inter-domain paths in the internet often fulfill a valley-free routing constraint [1,2] which is a simple condition that models real-life business agreements between different operators. Compared to a policy-free routing model, these conditions severely restrict allowed paths and might have a large impact on the overall throughput between two nodes when the conditions are strictly enforced. Conditions also influence path diversity, which in turn has a large impact on the overall resiliency of the connection.

Calculating the maximum throughput that can be achieved between two nodes in a network can easily be done via classic techniques solving the maximum flow problem, such as the algorithm of Ford–Fulkerson or the push-relabel method [3,4]. However, these algorithms do not take into account policies and do not enforce any condition or constraint at all. In this work, we will adapt generic flow algorithms, such that the flow that is achieved actually is a policy-compliant maximum flow. Policies are defined using finite state automata (FSA) [5,6], as their expressive power is sufficient for many purposes in real life, while still being conceptually simple.

We will define an algorithm that exactly solves the policy-compliant maximum flow problem, however, at the expense of a large computational footprint. Then, we will continue to adapt the classic

algorithms and datastructures such that they take into account policies that can be expressed using FSA. These algorithms, however, do not guarantee exact solutions to the problem anymore; they merely provide lower bounds to the exact answer. At each step, we are able to tighten the lower bound, such that it comes closer to the exact solution. However, each step also implies additional computational work, so we are trading off the quality of the results with the time needed to run the algorithms.

The rest of the paper is organized as follows. First, in Section 2 we provide some background about policies and how they can model certain conditions and constraints that are applied in real life. We also point out some difficulties that arise when trying to adapt the classic algorithms. Then, in Section 3 we discuss the formal model and the classic algorithms from literature, and how we can formally model policy-compliant connections. Next, in Section 4 we adapt the algorithms that were discussed in Section 3 to obtain a series of techniques that allow us to calculate lower bounds to the problem. Finally, Section 5 discusses the results obtained, and concludes this work, after which we provide an outlook to future work.

2. Background and Context

Policies are very important in today's internet. Specific agreements between operators are often hidden from prying eyes, but they can have a huge impact on how specific paths are set up between them. It turns out that many network policies, used in practice, can be translated in very simple formal languages, all belonging to a class of formal languages called "regular languages" [5,6]. This class is one of the most basic classes of languages used in computer science, as it is non-trivial but contains a very broad range of languages with widely differing properties that are very useful in practice.

The complexity of properties which we can express using regular languages is larger than would be expected at first sight. To provide a non-trivial, albeit contrived example, consider a government that wants to contact its embassy located in another country. Some links are trusted, as they are operated by friendly states, but some links in between may be eavesdropped upon by a rival state. Traffic crossing such links needs to be encrypted; once encrypted, traffic may be routed through any link in the network. The receiving party will need to decrypt the message, before it can be distributed to internal data-processing departments. The government thus places into force the following conditions: traffic from the government to the embassy should either be sent through links owned exclusively by friendly states, or it should pass the encryption box, after which it can be routed through any link. However, if encrypted, the message has to be decrypted before delivery at the internal departments at the embassy, i.e., pass through a specific decryption box.

It is clear that enforcing such policies is important, moreover such policies should be formalized instead of specified in vague terms as above. Regular languages fit that job quite well, as they allow policies like the one above, next to other much more involved policies, to be specified without any possibility for misunderstanding. Moreover, once specified, network-routes can be automatically validated against a set of policies, and compliance can be trivially checked.

Taking into account policies in the network, a natural question is to ask for their impact on the routing between nodes in the network. More specifically, in this work we are mainly interested in their impact on the maximum flow achievable between two nodes. Indeed, previously acceptable paths suddenly do not obey the specific policies, and as such, some routes, maybe all, will become invalid. Thus, the policy-compliant maximum flow between two nodes needs to be redefined and recalculated. Gaining insight into the flow-structure between two nodes also tells us something about resiliency in the network. Operators might be convinced their network has sufficient spare capacity, however, depending on the policy at hand and the specific structure of the network, a failing link might have tremendous consequences as re-routing traffic around the failure might invalidate the policy which is unacceptable.

We thus come to the following main problem-statement that will be treated in this work: how to calculate or approximate the policy-compliant maximum flow between two nodes?

Clearly, policy-compliance and both the theoretical and practical consequences are important in the design and operation of networks. From a practical point of view, much information can be found about policies and being compliant, and multiple frameworks for the management and monitoring of policies exist and are in current use. In contrast, few theoretical models have been developed, and literature is sparse when taking a more fundamental approach to the problem.

Caesar and Rexford [7] discuss how routing policies came into existence, and how the protocols have evolved over time and became increasingly complex. They discuss how common policies are implemented and address the problems that arise in applying and supporting policies. Feamster et al. [8] discuss fundamental objectives for interdomain routing and traffic engineering and provide practical guidelines. They show how greater flexibility can be gained in several situations and demonstrate the manipulation of traffic via small changes in specific routes. The paper by Hu et al. [9] proposes an approach to overcome the inherent constraints of compliant recovery schemes. Adapting protocols, they succeed in improving route diversity, in turn increasing resilience.

Klöti et al. [10] proposed a graph-transformation technique that constructs a tensor product of a graph and a finite state automaton. They show how to model policies using FSA and apply standard flow maximization algorithms on the transformed graph in order to obtain bounds for the policy-compliant maximum flow problem. Sobrinho et al. [11] provide a deep mathematical analysis to gain insight in the inner workings of route-vector protocols. They relate this to a class of routing policies and quantify how much intrinsic connectivity is lost due to typical routing policies. Erlebach et al. [12] approximate the maximum number of edge- and node-disjoint valid paths between two nodes, via an involved mathematical theory and specific approximation-algorithms.

Soulé et al. [13] provide a declarative formal language based on logical formulas and regular expressions to express network policies in the Merlin framework. After compilation, a constraint solver allocates paths. Tools are provided to verify whether conditions are violated. Hinrichs et al. [14] introduce a declarative policy management language. They focus on expressive power so that policies can be expressed naturally while still being able to enforce policies efficiently. They focus on enterprise networks, and their design using formal mathematical languages is attractive.

Raghavan et al. [15] provide a practical authenticated source routing system that allows for fine-grained path selection and enforcement of cryptographic policies. Capabilities can be defined and composed resulting in complex statements defining specific policies.

Godfrey et al. [16] introduce so-called pathlet routing, pathlets being building blocks which can be concatenated to obtain end-to-end routes fulfilling policy constraints. Pathlet routing can handle typical policies, but also other recent multipath routing proposals or source-routing approaches. Batista et al. [17] propose a policy-based OpenFlow network management framework using the Ponder policy definition language. They focus on simplicity of the system, trying to infer both static and dynamic conflicts. In [18] they update their work and propose a more theoretical approach to conflict detection using first-order mathematical logic to model flows. A Prolog rule-engine applies condition-action rules to infer problems. Xu and Rexford [19] discuss problems concerning multipath interdomain routing and introduce MIRO (Multi-path Interdomain ROuting), which gives transit domains control over the flow across their infrastructure. MIRO remains backwards compatible with other technologies and offers large flexibility with reasonable overhead.

3. Graph Model and Basic Algorithms

We now introduce basic concepts that are needed in this work. Intuitively, we can think about a flow network like an oil-producing plant, where oil is drilled at one site and needs to be transported to an oil depot via a set of pipes. Following the flow from the source to the sink, the oil can be split or joined at junctions. Most important here is the fact that the path that was followed by a certain molecule of oil is of no importance: it carries no history and passes through junctions anonymously. In contrast, in this work, we need to pay attention to the specific path that a certain amount of flow has followed: it carries its history with itself, see Figure 1. The amount of history that accompanies

the flow is, however, limited: it is represented via its state, and the number of possible different states is limited by the size of the automaton (cf. infra). To understand how to model policies using FSA, we refer to [10].

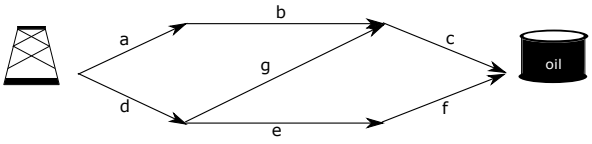

Figure 1. It doesn't matter for oil whether it followed path abc, path def or path dgc, but in this paper, we do care about the specific paths followed. Policies about which path is acceptable and which path is not, are expressed via constraints on the labels across the paths, i.e., abc, def and dgc.

3.1. Flow Networks and Flows

We now introduce flow networks and flows, following the approach taken by Cormen et al. [4]. These definitions might differ from other approaches in several aspects, e.g., the fact that the capacity function c and the flow f are total functions, allowing more concise and elegant definitions of constraints. The rationale behind this way of defining the context is included in Cormen et al. [4] and we refer interested readers to study the mathematical discussions in that book. We thus define a flow network $G = (V, E)$ as a directed graph together with a capacity function $c : V \times V \to \mathbb{R}$ such that each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. Note that c is a total function such that, if $(u, v) \notin E$, we have $c(u, v) = 0$. Moreover, we require that if E contains a certain edge (u, v), then there is no reverse edge (v, u) present in E. We also disallow loops such that $\forall u : (u, u) \notin E$. Two specific nodes in the graph are now chosen out of V, one vertex being the source s and one vertex being the sink t. We now want to send a flow from s to t, a flow being a total function $f : V \times V \to \mathbb{R}$ subject to the following constraints:

- A capacity constraint which expresses that flow can never exceed capacity:

$$\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v) \tag{1}$$

- A flow conservation constraint which expresses that for all nodes (except source and sink) the incoming flow equals the outgoing flow:

$$\forall u \in V - \{s, t\} : \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \tag{2}$$

Note that $f(u, v) = 0$ if $(u, v) \notin E$.

The value $|f|$ of a flow is defined as the flow leaving the source minus the flow arriving at the source: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Equivalently, we can define the value of the flow just as well as $|f| = \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v)$. Note that one can define the value of the flow using a formula containing only s but not t, or a formula containing only t but not s; one never needs both s and t in one formula at the same time, see the book by Cormen et al. [4]. Also note that we allow $\sum_{v \in V} f(v, s)$ as well as $\sum_{v \in V} f(t, v)$ to be strictly larger than 0. The classic problem, given a flow network G, is to maximize the flow $|f|$ from s to t under the constraints above. (The $|.|$-notation denotes flow value, not absolute value or cardinality.)

3.2. Residual Graphs and Augmenting Paths

One line of solving this so-called maximum flow problem is via the introduction of residual graphs and the calculation of augmenting paths. In short, we iteratively increase the value of the flow, starting with $\forall u, v \in V : f(u, v) = 0$. At each step, we calculate an augmenting path in the residual graph, which is nothing but the original graph, extended with return-edges. We will now formally define the necessary concepts.

First, we define the residual capacity $c_f : V \times V \to \mathbb{R}$ in the following way. We iterate over all edges $(u, v) \in E$ and for each such edge we define the following:

$$\begin{cases} c_f(u,v) = c(u,v) - f(u,v) \\ c_f(v,u) = f(u,v) \end{cases} \tag{3}$$

If there is no edge between u and v in any direction, i.e., $(u, v) \notin E$ and $(v, u) \notin E$, then both $c_f(u, v) = 0$ and $c_f(v, u) = 0$.

Next, we define the residual network $G_f = (V, E_f)$, where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$. So, for each edge in the original graph we have a residual edge if the capacity of that edge is not filled up yet, and moreover we add a reverse edge when it carries any flow at all.

This residual graph will now be used to find augmenting paths. Given a flow network $G = (V, E)$ and a flow f, an augmenting path p is defined as a path from s to t in the residual network, that is $p = [e_1, e_2, \ldots, e_n]$, such that

- edges belong to E_f, thus
$$\forall e_i : e_i \in E_f, \tag{4}$$
- we start in s, thus
$$\exists v : e_1 = (s, v), \tag{5}$$
- succeeding edges are connected, thus
$$\forall i \in \{1, \ldots, n-1\} : \exists u, v, w : e_i = (u, v) \wedge e_{i+1} = (v, w) \tag{6}$$
- we end in t, thus
$$\exists v : e_n = (v, t). \tag{7}$$

In order to increase the flow f, we will change the flow along the edges of the augmenting path p with the maximum amount possible, namely the residual capacity of the path p which is

$$c_f(p) = min\{c_f(u, v) : (u, v) \in p\}. \tag{8}$$

The flow f can now be increased to a new flow f' as follows:

$$f'(u,v) = \begin{cases} f(u,v) + c_f(p) & \text{if } (u,v) \in p, \\ f(u,v) - c_f(p) & \text{if } (v,u) \in p, \\ f(u,v) & \text{otherwise.} \end{cases} \tag{9}$$

Iteratively finding augmenting paths in a residual graph and increasing the flow will finally come to a stop when there are no augmenting paths anymore. Then, the maximum flow has been reached. Depending on the procedure used to find augmenting paths, the above algorithm is called Ford–Fulkerson or Edmonds–Karp.

3.3. Realization of a Flow

Apart from calculating the value of the maximum flow in a flow network, we are also interested in the actual paths followed by the flow; this set of paths is called the "realization" of the flow. Augmenting paths are a device to calculate the maximum flow, however, augmenting paths are not real paths in the flow network as they might comprise edges from E_f that are not in E. Define a path p

from s to t in G in exactly the same way as an augmenting path, except that edges in the path should be contained in E instead of E_f. Thus, in addition to (5)–(7), we replace (4) with

$$\forall e_i : e_i \in E. \tag{10}$$

Given a flow network G and accompanying maximum flow f, it is possible to determine a set

$$S = \{(p_1, c_1), (p_2, c_2), \ldots, (p_n, c_n)\} \tag{11}$$

that realizes that flow. Intuitively, we start with no flow in the network at all. Then, for each of the paths p_i we fill all edges of that path with an additional amount c_i. After all paths are processed, we obtain the maximum flow f.

Construction of this set S can be done simultaneously with the calculation of the maximum flow. During execution of the maximum flow algorithm, we perform the following steps every time a new augmenting path p with additional flow amount c has been found, starting with an empty set $S = \{\}$.

1. We split the augmenting path in forward sections and backward sections. That is, if $p = [e_1, e_2, \ldots, e_n]$, we cut it down and obtain $[[e_1, \ldots, e_i], [e_j, \ldots, e_k], \ldots, [e_l, \ldots, e_n]]$ such that every section $[e_x, \ldots, e_y]$ contains only edges from E or only edges from $E_f - E$. Without losing generality, we only discuss the case where the augmenting path p consists of two forward sections with one backward section in between, thus $p = [[e_1, \ldots, e_i], [e_j, \ldots, e_k], [e_l, \ldots, e_n]]$ such that $\{e_1, \ldots, e_i, e_l, \ldots, e_n\} \subseteq E$ and $\{e_j, \ldots, e_k\} \subseteq E_f - E$.
2. For this backward section $[e_j, \ldots, e_k]$ there must exist a path p' carrying some flow in the opposite direction, which the augmenting path p is canceling out. Indeed, as $[e_j, \ldots, e_k]$ are edges in E_f, they have been introduced in the residual graph G_f thanks to the fact that the corresponding forward edges in E have already been used in a previously found path. Thus, if $e_j = (u_j, v_j)$ and $e_k = (u_k, v_k)$, then S will already contain an item (p', c') where the path $p' = [\ldots, (v_x, v_k), (v_k, u_k), \ldots, (v_j, u_j), (u_j, u_y), \ldots]$, as the augmenting path p is crossing that section in the reverse direction, see Figure 2.
3. Remove this item (p', c') from S, and add two new paths $[e_1, \ldots, e_i, (u_j, u_y), \ldots]$ and $[\ldots, (v_x, v_k), e_l, \ldots, e_n]$, each with amount c. As such, the augmenting path has been cut and glued together with a previously found path, resulting in two new paths.
4. Now, it might be that $c < c'$. In that case, path p' is not to be removed from S entirely, as it still carries some flow that was not canceled by p. Add $(p', c' - c)$ to S.

Thus, each iteration of the maximum flow algorithm results in an augmenting path that is possibly cut in sections to remove reversed edges from the residual graph. Forward sections are glued together with sections from previously found paths. When no augmenting paths can be found, the set S will contain paths and associated flow amounts, that together sum up to the maximum flow. That is, if upon termination $S = \{(p_1, c_1), \ldots, (p_n, c_n)\}$, we have

$$f(u, v) = \sum_{\{(p_i, c_i) : (u,v) \in p_i\}} c_i \tag{12}$$

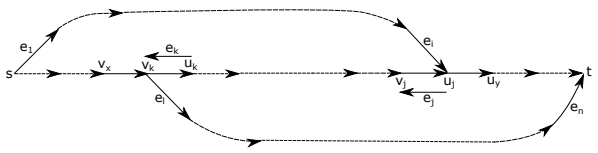

Figure 2. Three sections: forward, backward and forward.

Thanks to the fact that every flow has a realization and the fact that it is easy to derive the flow from a realization, from now on we identify a realization of a flow and the flow itself, and use the two concepts as one.

4. Policy-Compliant Paths and Algorithms

We now continue to define the concept of policy-compliant paths. Then we want to find the maximum flow in the network, allowing only the use of such policy-compliant paths. We will provide algorithms that calculate lower bounds, as well as exact solutions.

4.1. Finite State Automata

We now proceed to define finite state automata [5,6], which are the simplest of all classic automata. Note that we will only consider deterministic automata, although for finite state automata we have that non-determinism does not add expressive power, i.e., the class of deterministic finite state automata is exactly the same as the class of non-deterministic finite state automata.

We define a deterministic finite state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of internal states,
- Σ is a finite set of symbols, called the input alphabet,
- $\delta : Q \times \Sigma \to Q$ is the (total) transition function,
- $q_0 \in Q$ is the start (or initial) state,
- $F \subseteq Q$ is the set of final states.

Such an automaton works as follows. Initially, we start with an input word $w \in \Sigma^*$, where Σ^* is the set of strings obtained by concatenating zero or more symbols from Σ. Each of these symbols of the word w gets consumed, from left to right, while the automaton changes its state accordingly to its transition function, taking as input the current state and the symbol just read. If the automaton starts in state q_0, and finishes reading the word w while reaching state q, then we say that the word w is accepted by the automaton if and only if $q \in F$. For convenience, we introduce the extended transition function $\delta^* : Q \times \Sigma^* \to Q$, where the second argument is a string rather than a single symbol. We define δ^* recursively as follows:

$$\delta^*(q, w) = \begin{cases} q & \text{if } w = \lambda, \\ \delta^*(\delta(q, a), v) & \text{if } w = av, \end{cases} \tag{13}$$

where $a \in \Sigma$ and $v, w \in \Sigma^*$. Note that we also introduced the empty word λ out of convenience for this definition. We won't need it any further in this work.

4.2. Labeled Networks and Policy-Compliant Paths

Given a certain flow network $G = (V, E)$ and an FSA M, we define a labeling function $l : E \to \Sigma$ that attaches one symbol to every edge of the graph, which becomes a labeled flow network. Before, we were interested in all paths from source to sink, in order to maximize the total flow value of the network. Now, we constrain the allowed paths to only those that are accepted by the FSA M. Formally, we define a policy-compliant path p from s to t in G to be a sequence of edges from E, such that the symbols that are encountered, in order, constitute a word that is accepted by M. Thus, in addition to (5)–(7) and (10), we add the following constraint:

$$\delta^*(q_0, l(e_1)l(e_2)\dots l(e_n)) \in F \tag{14}$$

Thus, we are looking for paths from source to sink that are accepted by the automaton. Now we can define the policy-compliant maximum flow problem (PCMF) as calculating the maximum flow $|f|$

from s to t in a labelled flow network G_f, subject to the constraint that all paths that realize the flow f are accepted by an FSA M, i.e., they are policy-compliant paths.

4.3. Brute Force Approach

A basic, inefficient way of solving the policy-compliant maximum flow problem is by way of brute force. In its essence, we first calculate all policy-compliant paths from source to sink, and afterward try out all different orders in which these paths can be filled up. The maximum flow will be reached by at least one of these orderings. As such, this approach will result in the exact solution, however, at the expense of large computational complexity and memory requirements. Indeed, as we employ our search via e.g., a depth-first-search, it might be the case that we need to visit a certain vertex $u \in E$ more than once, because the internal state of the FSA might be different. Effectively, we need to build a new graph G_{BF} that contains tuples of vertices and states. After appropriately defining the edges between these tuples, a typical depth-first-search might be used to enumerate successively all policy-compliant paths. More formally, we perform the following steps.

1. Define G_{BF} as the graph with $V_{BF} = \{(u, q) : u \in V \wedge q \in Q\}$ and $E_{BF} = \{((u_1, q_1), (u_2, q_2)) : u_1, u_2 \in V \wedge q_1, q_2 \in Q \wedge (u_1, u_2) \in E \wedge \delta(q_1, l((u_1, u_2))) = q_2\}$. That is, the vertices of G_{BF} are all possible tuples of vertices in V and states in Q, and the vertices in G_{BF} are connected if and only if the corresponding vertices u_1 and u_2 in V were connected by an edge $e \in E$, such that the FSA changes its state from q_1 to q_2 when reading the symbol $l(e)$ associated with the original edge. Thus, we are defining G_{BF} as the Cartesian product of V and Q.
2. Start a depth-first-search from (s, q_0) to all (t, q) such that $q \in F$. This will produce all paths of the form $p = [e_1, \ldots, e_n]$ with $\forall e_i : e_i \in E_{BF}$ while $\exists v \in V_{BF} : e_1 = ((s, q_0), v)$ and $\exists v \in V_{BF} : \exists q \in F : e_n = (v, (t, q))$. Every such path corresponds to a policy-compliant path in G, and every policy-compliant path in G will be found this way. Collect all paths found by this depth-first-search in the set $BF = \{p_1, \ldots, p_n\}$.
3. Generate all permutations of the elements in this set BF; this results in $n!$ sequences of the n paths. Each of these sequences will correspond to an ordering in which we fill the paths to their residual capacity at that time. Note that it is not necessary to generate all permutations at once. It is sufficient to do this iteratively via e.g., the Steinhaus–Johnson–Trotter algorithm.
4. Each such sequence $s = [p_1^s, \ldots, p_n^s]$ contains exactly the same items as BF itself. We define an empty flow $f_s(u, v) = 0$ and iterate through this sequence: for each path $p_i^s = [e_{i,1}^s, \ldots, e_{i,m}^s]$ we calculate $min_i^s = min\{c(e_{i,j}^s) - f_s(e_{i,j}^s) : 1 \leq j \leq m\}$ which represents the largest amount of flow that can be accommodated along the path p_i^s at that moment. The flow f_s can now be increased to a new flow f_s' as follows:

$$f_s'(u, v) = \begin{cases} f_s(u, v) + min_i^s & \text{if } (u, v) \in p_i^s, \\ f_s(u, v) & \text{otherwise.} \end{cases} \tag{15}$$

5. After all paths p_i^s are processed, the final flow f_s will be a lower bound for the policy-compliant maximum flow for the network G. Thus, $max\{|f_s|\}$ will be the value of the policy-compliant maximum flow.

It is clear that this approach results in the exact value of the solution to the policy-compliant maximum flow problem. However, except for small networks and automata it will quickly take too much space and time to execute.

4.4. First Lower Bound

To cut down on the needed computational resources, we start from the classic Ford–Fulkerson or Edmonds–Karp algorithm. In its essence, these algorithms start with an empty flow, after which a new augmenting path is looked for in the residual graph. If found, this augmenting path gives rise to

an augmented flow, after which the residual graph is updated. Then, a new iteration is started. Once no new augmented paths can be found, the algorithm has reached the maximum flow.

In order to apply this approach to the policy-compliant maximum flow problem, it will be necessary to keep track of how the flow actually is realized, throughout the execution of the algorithm. This relates to the fact that flows cannot merge and split anonymously anymore, as discussed previously.

As a first step, it will be necessary to keep track of the different state-transitions that occur when flow crosses an edge $e = (u, v) \in E$ in a path p and changes state during that transition from state $q_u \in Q$ in vertex u to state $q_v \in Q$ in vertex v. To that aim, we define transitions t to be members of the set $Q \times Q = \{(q_u, q_v) : q_u \in Q \wedge q_v \in Q\}$. For every edge $e \in E$ we will now keep track of all transitions that cross that edge, and thus define the total function $T : V \times V \times Q \times Q \to \mathbb{R}$. This function keeps track of the amount of flow that crosses the edge (u, v) while changing state from q_u to q_v as the value $T(u, v, q_u, q_v)$. As a direct consequence, we have at all times $\forall u, v \in V : \sum_{q_u, q_v \in Q} T(u, v, q_u, q_v) = f(u, v) \leq c(u, v)$ which expresses the fact that a flow over an edge can be split up in different transitions, the amount of which add up exactly to the amount of flow crossing that very edge.

Keeping track of this function T allows the algorithm to discriminate between the different states in one single vertex, and the amount of the flow that is actually passing through that vertex having that state at that moment. This information is then used to glue together already found policy-compliant paths and a newly found augmenting path. In detail, this procedure works as follows.

1. Beginning with a labeled network G and a FSA M, we start a new flow and set $f(u, v) = 0$ for all vertices u and v. Moreover, we set $T(u, v, q_u, q_v) = 0$ for all vertices $u, v \in V$ and all states $q_u, q_v \in Q$.
2. Finding the first policy-compliant augmenting path can be straightforwardly done. Indeed, as the residual graph G_f is still the same as the original graph G (because $\forall u, v : f(u, v) = 0$), the augmenting path that is found in the first iteration will be a path containing only forward sections. The flow is then updated according to (9).
3. From the second iteration on, we need to take into account the reverse edges in the residual graph with great care. Now, it is not sufficient anymore to simply look for policy-compliant paths containing only forward sections, as an augmenting path may now contain reverse edges from the residual graph. In general, we will look for an augmenting path $p = [e_1, \ldots, e_n]$ where $e_i \in E_f$, that can have multiple forward and backward sections, and which can thus be split as $p = [[e_1, \ldots, e_i], [e_j, \ldots, e_k], \ldots, [e_l, \ldots, e_n]]$, such that each section $[e_x, \ldots, e_y]$ only contains edges from either E or $E_f - E$. We will now determine the constraints that are to be applied to ensure that the newly found augmenting path p can be split up and glued together with other policy-compliant paths that are already part of the flow f.
4. For simplicity, we discuss the case where the augmenting path p consists of two forward sections with one backward section in between, that is $p = [[e_1, \ldots, e_i], [e_j, \ldots, e_k], [e_l, \ldots, e_n]]$ such that $\{e_1, \ldots, e_i, e_l, \ldots, e_n\} \subseteq E$ and $\{e_j, \ldots, e_k\} \subseteq E_f - E$. To identify the vertices involved, suppose that $e_j = (u_j, v_j)$ and $e_k = (u_k, v_k)$, so $p = [[e_1, \ldots, e_i], [(u_j, v_j), \ldots, (u_k, v_k)], [e_l, \ldots, e_n]]$, see Figure 2. Starting from state q_0, the start state of the automaton, we will reach state $q_{u_j} = \delta^*(q_0, l(e_1) \ldots l(e_i))$ after we crossed all edges $[e_1, \ldots, e_i]$ in the first forward section of the augmenting path.
5. The first forward section of the augmenting path that we have found will be cut at this point, at the cut-vertex u_j. For this first approach to obtain a lower bound for the PCMF problem, we assert the overly restrictive constraint that $\exists u_y \in V$ and $\exists q_{u_y} \in Q$ such that

$$(u_j, u_y) \in E \wedge T(u_j, u_y, q_{u_j}, q_{u_y}) > 0. \tag{16}$$

Choose such an edge $e = (u_j, u_y) \in E$ and state $q_{u_y} \in Q$. When the augmenting path p is cut at vertex u_j, we can glue the new flow that comes in via the forward section $[e_1, \ldots, e_i]$ to the flow that leaves vertex u_j in state q_{u_j} via edge $e = (u_j, u_y)$ and reaches state q_{u_y} at the other end in vertex u_y. This flow is already available and does not concern us anymore.

6. Now, we need to cancel flow that used to cross the edges $[(v_k, u_k), \ldots, (v_j, u_j)]$, that is the backward section of the augmenting path p, crossed in the opposite direction. This gives rise to a second condition for this algorithm to find augmenting paths. This section contains a flow, reaching state q_{u_j} at the end. Tracing back where this flow comes from, we require that there must exist some flow passing vertex v_k in state $q_{v_k} \in Q$ which crosses the edges $[(v_k, u_k), \ldots, (v_j, u_j)]$ finally reaching vertex u_j in state q_{u_j}. This observation immediately translates to the following condition:

$$\exists q_{v_k} \in Q : \delta^*(q_{v_k}, l((v_k, u_k)) \ldots l((v_j, u_j))) = q_{u_j}. \tag{17}$$

Choose such a $q_{v_k} \in Q$. Then, the flow that is already present in the network and passes vertex v_k in state q_{v_k} will be canceled over the section $[(v_k, u_k), \ldots, (v_j, u_j)]$.

7. We now need to consider the last forward section $[e_l, \ldots, e_n]$. This section starts in vertex v_k and ends in the sink t, thus $\exists v \in V : e_l = (v_k, v)$ and $\exists v \in V : e_n = (v, t)$. We know that, having reached state q_{v_k} in v_k, we will cross edges $[e_l, \ldots, e_n]$ and need to reach some state $q_t \in F$ to obtain a policy-compliant augmenting path. This immediately gives rise to the third and last condition, which can be stated as

$$\delta^*(q_{v_k}, l(e_l) \ldots l(e_n)) = q_t \in F. \tag{18}$$

8. Suppose all conditions (16)–(18) for this augmenting path are fulfilled, which means that we have found an augmenting path that can be joined with the already available policy-compliant paths to increase the total policy-compliant flow. We now need to calculate the amount of flow that can be sent over this augmenting path. First, define $c_{u_j} = min\{c_f(u, v) : (u, v) \in [e_1, \ldots, e_i]\}$ to be the capacity that is at most available along the first forward section of the augmenting path and thus at the endpoint u_j of this section. Likewise, define $c_{v_k} = min\{c_f(u, v) : (u, v) \in [e_l, \ldots, e_n]\}$ to be the amount of flow that is at most available along the second forward section of the augmenting path, and thus the maximum amount of flow that can be diverted at the starting point v_k of this section. For the backward section of the augmenting path, define $c_{v_k u_k} = T(v_k, u_k, q_{v_k}, q_{u_k}), \ldots, c_{v_j u_j} = T(v_j, u_j, q_{v_j}, q_{u_j})$, to determine the already crossing flow over the backward edges, taking into account the state-transitions which we derived above. Taking everything together, we have

$$c_f(p) = min\{c_{u_j}, c_{v_k u_k}, \ldots, c_{v_j u_j}, c_{v_k}\} \tag{19}$$

(cf. (8)). This denotes the additional flow $c_f(p)$, under the assumption that $c_f(p) > 0$, that can now be sent over the newly found augmenting path p, such that the resulting flow f can be realized via only policy-compliant paths.

9. We now want to update the value of the flow f. This can be done via (9). However, we also need to update the specific values of the function T, for all edges involved in the augmenting path. First, we update T for all edges $e = (u_x, u_y)$ in the forward section $[e_1, \ldots, e_i]$ via $T'(u_x, u_y, q_{u_x}, q_{u_y}) = T(u_x, u_y, q_{u_x}, q_{u_y}) + c_f(p)$. Thus, for such an edge $e = (u_x, u_y)$ we add an additional $c_f(p)$ units of flow that change state from q_{u_x} to q_{u_y} while crossing edge e. For all edges $[(v_k, u_k), \ldots, (v_j, u_j)]$, which are crossed in the reverse direction in the backward section of the augmenting path, we set $T'(u_x, u_y, q_{u_x}, q_{u_y}) = T(u_x, u_y, q_{u_x}, q_{u_y}) - c_f(p)$, actually canceling out the pre-existent transitions in that section. Finally, for the edges in the final forward section $[e_l, \ldots, e_n]$ we update the transitions via $T'(u_x, u_y, q_{u_x}, q_{u_y}) = T(u_x, u_y, q_{u_x}, q_{u_y}) + c_f(p)$. This concludes the updates of

the transitions-function T and the work that needs to be done when an augmenting path has been found.

10. Iterate steps (3)–(9), that is, keep searching for augmenting paths that fulfill conditions (16)–(18) and update the flow f according to (9) and the transition-function T via the equations above. Once no more augmenting paths are found, stop iterating.
11. The resulting flow f can be realized via a set $\{p_1, \ldots, p_n\}$ of n policy-compliant paths p_i. Thus, the value $|f|$ is a lower bound for the value of the policy-compliant maximum flow problem.

4.5. Second Lower Bound

Although the approach described above results in correct lower bounds for the policy-compliant maximum flow problem, we point out that the conditions (16)–(18) that are imposed upon augmenting paths are very restrictive. Indeed, it is easy to describe specific cases where the approach misses additional augmenting paths due to these conditions, which can easily be relaxed at the expense of additional computational work. The next approach, which will result in a second lower bound that is tighter than the first bound, starts by relaxing condition (16).

In its essence, this condition says that for an augmenting path, containing a backward section, it is necessary to glue the immediately previous forward section to a pre-existent policy-compliant path at the cut-vertex, via an exact match of the state in that cut-vertex. Clearly, this is too restrictive a condition as it would be sufficient if the augmenting flow arriving at that vertex could be continued via the same edges, however taking different intermediate states in between.

1. To make this formal, consider the augmenting path $p = [e_1, \ldots, e_n]$ where $e_i \in E_f$, which may have multiple forward and backward sections, and which can thus be split as $p = [[e_1, \ldots, e_i], [e_j, \ldots, e_k], \ldots, [e_l, \ldots, e_n]]$, such that each section $[e_x, \ldots, e_y]$ only contains edges from either E or $E_f - E$. For simplicity, we once more discuss the case where the augmenting path p consists of two forward sections with one backward section in between, that is $p = [[e_1, \ldots, e_i], [e_j, \ldots, e_k], [e_l, \ldots, e_n]]$ such that $\{e_1, \ldots, e_i, e_l, \ldots, e_n\} \subseteq E$ and $\{e_j, \ldots, e_k\} \subseteq E_f$. To identify the vertices involved, say that $e_j = (u_j, v_j)$ and $e_k = (u_k, v_k)$, so $p = [[e_1, \ldots, e_i], [(u_j, v_j), \ldots, (u_k, v_k)], [e_l, \ldots, e_n]]$. The state at u_j is defined as $q_{u_j} = \delta^*(q_0, l(e_1) \ldots l(e_i))$.
2. For flow f, suppose that the policy-compliant paths $P = \{p_1, \ldots, p_n\}$ are the part of the realization of f that crosses u_j. Thus, $\forall p_i \in P : \exists v_i \in V : (u_j, v_i) \in p_i$. For each of these policy-compliant paths p_i, the state upon arriving in u_j is denoted by $q^i_{u_j}$. Moreover, there might be more than one path $p_i \in P$ that arrives in u_j with state $q^i_{u_j}$, that is for some i and j it might be the case that $i \neq j$ and $q^i_{u_j} = q^j_{u_j}$. The amount of flow that policy-compliant path p_i carries is denoted by c_i, thus $\forall p_i \in P : (p_i, c_i) \in S$, cf. (11).
3. Now, instead of stating in the first condition (16) that there must be a policy-compliant path $p_i \in P$ such that $q_{u_j} = q^i_{u_j}$ with $c_i > 0$ it is, in fact, sufficient that there exists a path $p_i \in P$ such that following the rest of the policy-compliant path p_i after crossing u_j results in a final state at the sink t even though $q_{u_j} \neq q^i_{u_j}$. More formally, if $p_i = [e^i_1, \ldots, e^i_k, e^i_l, \ldots, e^i_n]$ such that $e^i_k = (u_x, u_j)$ and $e^i_l = (u_j, u_y)$ for some $u_x \in V$ and $u_y \in V$, then it is sufficient to impose, instead of (16), simply that
$$\exists p_i \in P : \delta^*(q_{u_j}, l(e^i_l) \ldots l(e^i_n)) \in F. \tag{20}$$
Choose such a $p_i \in P$. That way, we are confident that the prefix $l(e_1) \ldots l(e_i)$ can be joined together with a suffix $l(e^i_l) \ldots l(e^i_n)$ such that for the whole word holds that $\delta^*(q_0, l(e_1) \ldots l(e_i) l(e^i_l) \ldots l(e^i_n)) \in F$.
4. At this point, we have to be careful which policy-compliant flows crossing the backward section $[e_j, \ldots, e_k]$ we are going to cancel. Indeed, as the first section $[e_1, \ldots, e_i]$ is glued together with

some policy-compliant path $p_i \in P$ at vertex u_j via condition (20), we are looking for states $q_{v_k} \in Q$ at vertex v_k such that, instead of (17) the following formula holds:

$$\exists q_{v_k} \in Q : \delta^*(q_{v_k}, l((v_k, u_k)) \dots l((v_j, u_j))) = q^i_{u_j}. \tag{21}$$

Indeed, after crossing section $[e_j, \dots, e_k]$ in reverse direction we want to end in state $q^i_{u_j}$ instead of q_{u_j}.

5. Having fulfilled the condition to allow canceling flow across the backward section of the augmenting path, we now need to check whether the flow arriving in v_k in state q_{v_k} can actually be diverted via the last forward section $[e_l, \dots, e_n]$ to the sink t reaching state q_t which needs to be a final state. This is accurately expressed by (18) which we can leave unmodified.
6. Suppose conditions (18), (20) and (21) have been fulfilled. Then calculate the residual capacity (19). Update flow f via (9) across the augmenting path using the value for the residual capacity, and update transitions-function T across the augmenting path as done previously.
7. Finally, we need to update the transitions-function T across the final part of the policy-compliant path p_i, cf. (20). This final part consists of the edges $[e^i_l, \dots, e^i_n]$, and thus we need to update T for all edges (u_x, u_y) in this partial path. We update T via the formulae $T'(u_x, u_y, q^i_{u_x}, q^i_{u_y}) = T(u_x, u_y, q^i_{u_x}, q^i_{u_y}) - c_f(p)$ and $T'(u_x, u_y, q_{u_x}, q_{u_y}) = T(u_x, u_y, q_{u_x}, q_{u_y}) + c_f(p)$, where we use the notation $q^i_{u_x}$ for the state that is reached at vertex u_x when starting in state $q^i_{u_j}$ in vertex u_j, while q_{u_x} denotes the state that is reached at vertex u_x when starting in state q_{u_j} in vertex u_j. After these updates, the augmenting path is processed entirely, and a new iteration can start.

Note that, in comparison to the approach for obtaining the first lower bound, condition (20) not only has merits for being more relaxed, it also carries a serious drawback as checking whether condition (20) holds involves more computational work than (16). One possible approach to check whether (20) actually holds, is by a depth-first walkthrough of the realization associated with the flow f. However, this is complicated by the fact that the flow might have multiple valid realizations, some of which are compatible with (20), while some are not. We discuss this now in detail.

1. Consider the augmenting path $p = [[e_1, \dots, e_i], [(u_j, v_j), \dots, (u_k, v_k)], [e_l, \dots, e_n]]$ from above, where the state at u_j is defined as $q_{u_j} = \delta^*(q_0, l(e_1) \dots l(e_i))$. Suppose there exist two policy-compliant paths $p_i \in P$ and $p_j \in P$, such that their states in vertex u_j are $q^i_{u_j}$ and $q^j_{u_j}$ respectively. Suppose that the realization of the flow that gave rise to these paths p_i and p_j is such that $p_i = [\dots, (u_j, u_1), (u_1, u_3), (u_3, u_4), (u_4, t)]$ and $p_j = [\dots, (u_j, u_2), (u_2, u_3), (u_3, u_5), (u_5, t)]$ and moreover that $l((u_j, u_1)) = a$, $l((u_1, u_3)) = b$, $l((u_3, u_4)) = c$, $l((u_4, t)) = d$, $l((u_j, u_2)) = e$, $l((u_2, u_3)) = f$, $l((u_3, u_5)) = g$ and finally $l((u_5, t)) = h$ with $\{a, b, c, d, e, f, g, h\} \subseteq \Sigma$. Then, as p_i and p_j are policy-compliant paths in the set P, it directly follows that $\delta^*(q^i_{u_j}, abcd) \in F$ and $\delta^*(q^j_{u_j}, efgh) \in F$, see Figure 3.

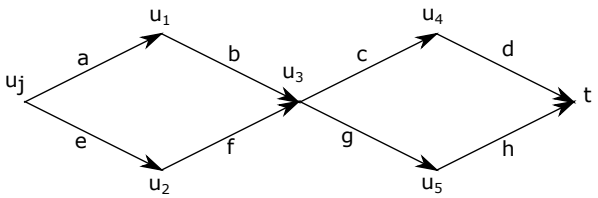

Figure 3. Fulfilling condition (20).

2. Suppose that moreover it holds that $\delta^*(q^i_{u_j}, abgh) \in F$ and $\delta^*(q^j_{u_j}, efcd) \in F$, that is, for the paths p_i and p_j it doesn't really matter which way they take in the second diamond. However, suppose that $\delta^*(q_{u_j}, abgh) \in F$ as well as $\delta^*(q_{u_j}, efcd) \in F$ but $\delta^*(q_{u_j}, abcd) \notin F$ and also $\delta^*(q_{u_j}, efgh) \notin F$.

3. Now, there is no way to fulfill condition (20), as no policy-compliant path through u_j that is part of the realization of f allows the prefix $l(e_1) \dots l(e_i)$ to be extended by a suffix to arrive at a final state in the sink. However, if the realization of the flow f was such that it contained the policy-compliant paths $[\dots, (u_j, u_1), (u_1, u_3), (u_3, u_5), (u_5, t)]$ and $[\dots, (u_j, u_2), (u_2, u_3), (u_3, u_4), (u_4, t)]$, then the prefix of the augmenting path p could have been extended via a matching suffix to a final state. In short, condition (20) would have been fulfilled.

Thus, although the relaxed condition (20) can be used to generate augmenting paths that would go unnoticed if applying (16), it suffers from the fact that previous decisions (like how to route policy-compliant paths in case of multiple possibilities) can have a large impact on the number of augmenting paths that can be found later on. As such, we have no guarantee that this second approach will always calculate the exact value for the policy-compliant maximum flow problem, but merely a lower bound. However, as (20) and (21) are fulfilled everytime (16) and (17) are fulfilled, we can immediately derive that the second lower bound will be tighter than the first lower bound.

4.6. Third Lower Bound

Considering the approaches taken to calculate the first and the second lower bound, observe that at several times choices have to be made, e.g., when fulfilling condition (17). Apart from the fact that some of the conditions might be overly restrictive, we face the fact that we might miss augmenting paths during the search when we make a bad choice, as well as the fact that we might need additional iterations to reach the point where no more augmenting paths can be found. For example, consider (17) and suppose that multiple states exist that fulfill the condition. By choosing one of them, an already present policy-compliant path is chosen too, which implies that the amount of flow that can be canceled has an upper bound that is equal to the minimum value of residual capacity across the backward section. In turn, this restricts the total amount of flow that can be sent over the augmenting path due to (19).

Thus, in order to allow as much flow as possible to be sent over the augmenting path that is constructed, one can check at the same time whether other transitions can be applied in parallel, such that the total amount of flow that is canceled across the backward section is maximized. First, we want to allow multiple pre-existing policy-compliant paths that cross the cut-vertex to be involved in the process, such that we do not have to choose at all which path we want to work with in (20). Multiple policy-compliant paths might fulfill the condition, even reaching the same state in the cut-vertex but following another way to the sink. Also, we want to allow multiple states at the end of the backward section to be involved in the process, such that we do not have to choose at all which state we want to work with in (21). Multiple states might fulfill the condition, and they even might be equal to each other even though they are part of different policy-compliant paths. We need to match the possibilities in both sets to maximize the flow that we can cancel along the backward section in the middle of the augmenting path. This being, of course, under the additional condition (18).

We start again with augmenting path p, set P and set S. Now select a subset $P' \subseteq P$ such that

$$\forall p_i \in P' : \delta^*(q_{u_j}, l(e_l^i) \dots l(e_n^i)) \in F \tag{22}$$

$$\forall p_i \in P' : \delta^*(q_{v_k}^i, l((v_k, u_k)) \dots l((v_j, u_j))) = q_{u_j}^i \tag{23}$$

$$\forall p_i \in P' : \delta^*(q_{v_k}^i, l(e_l) \dots l(e_n)) \in F \tag{24}$$

$$\Sigma_{p_i \in P'} c_i \leq min(c_{u_j}, c_{v_k}). \tag{25}$$

This set P' can be iteratively built, starting from the empty set and choosing and adding p_i's while (25) holds. Once the iteration comes to a stop, augment the flow along the path p with the value $\Sigma_{p_i \in P'} c_i$ and update f and T. As the conditions (18), (20) and (21) imply that (22)–(24) will also hold, we obtain once more a tighter bound. However, there is no guarantee that the policy-compliant maximum flow

will be reached. This is the most generic specification of a solution to the policy-compliant maximum flow problem that we derived.

4.7. Implementation and Experimental Results

Implementing classical maximum flow algorithms is not difficult as they are well-studied. Typical approaches include augmenting path algorithms, push-relabel algorithms and others. They have varying complexities in solving the problem. In practice, execution times are also dependent on the structure and density of the network. We chose an augmenting path approach. Finite state automata are typically implemented in a table-driven way, and it is quite straightforward to implement them efficiently, like we did. However, combining a maxflow algorithm with an FSA multiplies the complexities of both. Indeed, sending a flow over an edge multiple times was now allowed, as the states at the beginning and the end of the edge when crossing it for the first time might be different from the states while crossing it a second time. Thus, utmost care is needed to implement these algorithms correctly. All code is freely available in Supplementary Materials.

Although this paper is mainly intended as a theoretical study, we performed a large number of experiments validating our approach and formulas, in order to build some insights in the practical use of our approach, execution times and the accuracy reached. However, these values must be interpreted as rough estimates, and can only serve as rules-of-thumb, as much is dependent on the actual networks and finite state automata used in the experiments. Indeed, for evaluating the work, a network generator was implemented which allowed for the use of random networks having differing numbers of vertices and edges. Of course, the higher the density of edges, the more paths can be found and the higher the calculation time for the varying parts of the algorithms. We experimented with graphs having up to 256 vertices, with a random choice of edges (from one to the maximum number, i.e., $n(n-1)/2$). Also, we implemented an FSA generator, which allowed experimenting with different policies, all expressed as regular expressions. The maximal number of states allowed was maximally of the same order as the number of vertices, and clearly it holds that allowing more states leads to higher computational times. Indeed, a flow can reach a vertex in about $O(|V|)$ possible states, which also means that an algorithm looking for augmenting paths can be in about $O(|V|^2)$ states itself. Also, the flow over any edge is now parameterized by the states of both endpoints of the edge; as we use up to $|V|$ possible states this results in the memory-use for function T scaling in the order $O(|V|^4)$.

For our experiments with networks with up to 32 vertices and FSAs with up to 32 states, we found that running the brute-force algorithm either finished within a second, or did not finish even after 30 min, after which we always stopped the experiments. Careful investigation shows that the density of the network, combined with the structure of the FSA, is of prime importance to whether or not the algorithm finishes fast. For all the randomly generated experiments where the brute-force approach finished and provided us with the maximal value, the first lower bound also reached that same maximal value, but even in the worst observed cases in less than a millisecond, which is three orders of magnitude faster than the brute-force approach. The first lower bound algorithm could not find the maximal value only for specifically designed networks and FSAs, such as the network in Figure 3, aimed specifically at fooling the algorithm. For random experiments, the second lower bound algorithm always gives the same result as the first algorithm, taking only a little bit more time to execute. The third lower bound tries out much more combinations and comes closer to the brute-force approach in computational time, and in our random experiments always obtained the same maximal flow values as the second lower bound. We also executed random experiments with up to 256 vertices without having the maximal flow value, because the brute-force algorithm did not terminate. For the subset of these experiments where the three lower bound algorithms did terminate, also no differences were obtained in the resulting values. Thus, differences in the accuracy of results were never detected in any random experiment, and could only be obtained with specifically designed and contrived networks and automata. Runtimes for the first lower bound algorithm, for random networks having up to 256 vertices, are shown in Figure 4. If the algorithm finished, it typically finished fast (i.e., within

about 10 s), as can be seen in the figure. If it didn't finish, we aborted the execution after 30 min (not depicted).

Concluding, we advise to use the first lower bound algorithm, as it terminates very fast and for our random experiments always reached the maximal value when known, i.e., when the brute-force approach also finished. It couldn't find the maximal value only in contrived examples, specifically designed to fool the algorithm. Also, we did not encounter random experiments where the three algorithms gave different results, so there is no reasonable situation where one might recommend using the second or third algorithm. We would like to stress again that the execution times we obtained varied wildly, being highly dependent on the structure of the network, the accompanying FSA and the combinatorial combination of the two.

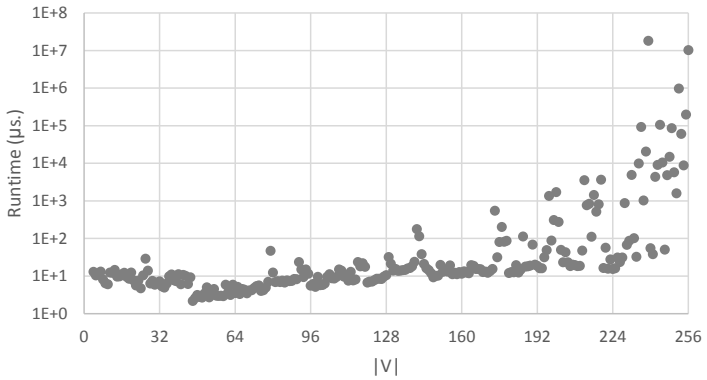

Figure 4. Runtimes for the first lower bound algorithm, for random networks with up to 256 vertices.

5. Conclusions and Future Work

Knowing the maximum throughput between two nodes in a network is key knowledge for a network operator. In practice, network policies severely complicate this task and often leave the question open-ended. We put forward a formal model with which it is possible to model real-life policy constraints, and we analyzed the impact of policies on throughput. As exact solutions are difficult to obtain, we defined a series of conditions and algorithms that allow us to calculate a sequence of increasingly tighter bounds on the exact value. These algorithms are built upon classic algorithms like Ford–Fulkerson to solve the generic maximum flow problem, which are adapted to our needs and augmented with specific functions to ease bookkeeping of additional data like transitions or the specific realization associated with the flow.

Although the approach to specify the conditions on the augmenting paths in order to enforce their policy-compliance is reasonable in combination with Ford-Fulkerson, it requires complex formulae that express conditions on the augmenting path, end-to-end. Future work in the direction of push-relabel algorithms would solve this problem: on the basis of local conditions, it might be possible to express specific constraints that guarantee path-compliancy upon arrival of the flow.

Supplementary Materials: All code is freely available and downloadable at http://www.dna.idlab.ugent.be.

Author Contributions: Conceptualization and methodology: P.A., D.C. and M.P.; software, validation, formal analysis, resources, visualization, writing: P.A.; investigation, writing—review and editing: P.A., D.C. and M.P.

Funding: This research was funded by Ghent University—imec, IOP 2016 Derudder-Pickavet.

Acknowledgments: The authors wish to thank Ghent University—imec for the support.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

FSA Finite state automaton
PCMF Policy-compliant maximum flow

References

1. Gao, L. On Inferring Autonomous System Relationships in the Internet. *IEEE/ACM Trans. Netw.* **2001**, *9*, 733–745. [CrossRef]
2. Gao, L.; Rexford, J. Stable Internet Routing Without Global Coordination. *IEEE/ACM Trans. Netw.* **2001**, *9*, 681–692. [CrossRef]
3. Ford, D.R.; Fulkerson, D.R. *Flows in Networks*; Princeton University Press: Princeton, NJ, USA, 2010.
4. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; The MIT Press: Cambridge, MA, USA, 2009.
5. Hopcroft, J.E.; Ullman, J.D. *Formal Languages and Their Relation to Automata*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1969.
6. Linz, P. *An Introduction to Formal Language and Automata*; Jones and Bartlett Publishers, Inc.: Burlington, MA, USA, 2006.
7. Caesar, M.; Rexford, J. BGP Routing Policies in ISP Networks. *IEEE Netw.* **2005**, *19*, 5–11. doi:10.1109/MNET.2005.1541715. [CrossRef]
8. Feamster, N.; Borkenhagen, J.; Rexford, J. Guidelines for Interdomain Traffic Engineering. *SIGCOMM Comput. Commun. Rev.* **2003**, *33*, 19–30. [CrossRef]
9. Hu, C.; Chen, K.; Chen, Y.; Liu, B. Evaluating Potential Routing Diversity for Internet Failure Recovery. In Proceedings of the 29th Conference on Information Communications, San Diego, CA, USA, 14–19 March 2010; IEEE Press: Piscataway, NJ, USA, 2010; pp. 321–325.
10. Klöti, R.; Kotronis, V.; Ager, B.; Dimitropoulos, X. Policy-Compliant Path Diversity and Bisection Bandwidth. In Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM), Kowloon, Hong Kong, China, 26 April–1 May 2015.
11. Sobrinho, J.L.; Quelhas, T. A Theory for the Connectivity Discovered by Routing Protocols. *IEEE/ACM Trans. Netw.* **2012**, *20*, 677–689. [CrossRef]
12. Erlebach, T.; Hall, A.; Panconesi, A.; Vukadinovic, D. Cuts and Disjoint Paths in the Valley-Free Model. *Internet Math.* **2007**, *3*, 333–359. [CrossRef]
13. Soulé, R.; Basu, S.; Kleinberg, R.; Sirer, E.G.; Foster, N. Managing the Network with Merlin. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, College Park, MD, USA, 21–22 November 2013; ACM: New York, NY, USA, 2013; pp. 24:1–24:7. [CrossRef]
14. Hinrichs, T.L.; Gude, N.S.; Casado, M.; Mitchell, J.C.; Shenker, S. Practical Declarative Network Management. In Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN '09), Barcelona, Spain, 21 August 2009; ACM: New York, NY, USA, 2009; pp. 1–10. [CrossRef]
15. Raghavan, B.; Verkaik, P.; Snoeren, A.C. Secure and Policy-compliant Source Routing. *IEEE/ACM Trans. Netw.* **2009**, *17*, 764–777. [CrossRef]
16. Godfrey, P.B.; Ganichev, I.; Shenker, S.; Stoica, I. Pathlet Routing. *SIGCOMM Comput. Commun. Rev.* **2009**, *39*, 111–122. [CrossRef]
17. Batista, B.; de Campos, G.; Fernandez, M. A proposal of policy based OpenFlow network management. In Proceedings of the 2013 20th International Conference on Telecommunications (ICT), Casablanca, Morocco, 6–8 May 2013; pp. 1–5. [CrossRef]
18. Lopes Alcantara Batista, B.; Lima de Campos, G.; Fernandez, M. Flow-based conflict detection in OpenFlow networks using first-order logic. In Proceedings of the 2014 IEEE Symposium on Computers and Communication (ISCC), Funchal, Portugal, 23–26 June 2014; pp. 1–6. [CrossRef]
19. Xu, W.; Rexford, J. MIRO: Multi-path Interdomain Routing. *SIGCOMM Comput. Commun. Rev.* **2006**, *36*, 171–182. [CrossRef]