Noname manuscript No.
(will be inserted by the editor)

Adversarial Topology Discovery in Network Virtualization Environments: A Threat for ISPs?

Yvonne Anne Pignolet · Stefan Schmid · Gilles Tredan

Abstract Network virtualization is a new Internet paradigm which allows multiple *virtual networks (VNets)* to share the resources of a given physical infrastructure. The virtualization of entire networks is the natural next step after the virtualization of nodes and links.

While the problem of how to embed a VNet ("guest network") on a given resource network ("host network") is algorithmically well-understood, much less is known about the security implications of this new technology. This paper introduces a new model to reason about one particular security threat: the leakage of information about the physical infrastructure—often a business secret.

We initiate the study of this new problem and introduce the notion of *request complexity* which describes the number of VNet requests needed to fully disclose the substrate topology. We derive lower bounds and present algorithms achieving an asymptotically optimal request complexity for important graph classes such as trees, cactus graphs (complexity $O(n)$) as well as arbitrary graphs (complexity $O(n^2)$). Moreover, a general motif-based topology discovery framework is described which exploits the poset structure of the VNet embedding relation.

This article is based on the conference publications [20–22].

Y.-A. Pignolet
ABB Corporate Research, Switzerland
E-mail: yvonne-anne.pignolet@ch.abb.com

S. Schmid
Telekom Innovation Laboratories & TU Berlin, Germany
E-mail: stefan@net.t-labs.tu-berlin.de

G. Tredan
LAAS-CNRS, France
E-mail: gilles.tredan@laas.fr

1 Introduction

Virtualization is arguably the main innovation motor in today's Internet. Already a decade ago, node virtualization revamped the server business, and today's datacenters host thousands of virtual machines. But also links become more virtualized, e.g., through the introduction of Software-Defined Networking [19] technology.

After the virtualization of *nodes* and *links*, the industry as well as the academia discusses the virtualization of entire *networks*: *network virtualization* [8,15] envisions an Internet where customers (e.g., a startup company or a content distribution provider) can request *virtual networks (VNets)* on short notice and with arbitrary specifications on the required node resources and their connectivity. Indeed, first prototypes have emerged (e.g., in the European CHANGE and UNIFY projects, the American GENI project or the Asian AKARI project). Network virtualization is particularly attractive for Internet Service Providers (ISPs): ISPs benefit from the improved resource utilization as well as from the possibility to introduce new services. [27]

This paper argues that the network virtualization trend also comes with certain threats. In particular, we show that critical information about the ISP's network infrastructure and its properties can be learned from answers to VNet embedding requests. Given that the resource network constitutes a competitive advantage and is a business secret, this is problematic; moreover, the discovery of, e.g., bottlenecks, may be exploited for attacks or bad publicity. Hence, providers around the world are often reluctant to open the infrastructure to novel technologies and applications that might lead to information leaks.

Background on VNets and Embeddings. Basically, a VNet defines a *graph* (henceforth: the *guest graph*): a set of virtual nodes (e.g., virtual machines)

which need to be interconnected via virtual links according to the specified VNet topology. This graph must be realized over a physical resource network (also called the *substrate network* or *host graph*): the virtual nodes must be mapped to physical nodes and the virtual links must be mapped to physical *paths* (e.g. realized by OpenFlow [19]).

In this paper we consider VNet requests which do not impose any location constraints on where the virtual nodes are mapped to: For example, a computational VNet to process large amounts of scientific data, a datacenter VNet, or a testbed VNet such as Planetlab to experiment with novel networking protocols may not specify locations. This flexibility in the VNet specification can be exploited to optimize the VNet embedding.

Although VNets appear as dedicated and "real" networks to their users, several VNets can be *embedded* (i.e., realized) over the same infrastructure network (referred to as the *substrate network*); network virtualization technology therefore enables resource reuse. In this paper, we will prefer the term *substrate* to the term *infrastructure* as the substrate network may not necessarily be a physical network, but a virtual network itself. [27]

Figure 1 illustrates the VNet embedding problem.

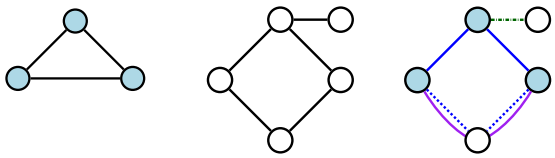

Fig. 1 In order to embed a VNet (i.e., a *guest graph* G) with unit link and node demands forming a triangle (*left*) in the substrate network (i.e., a *host graph* H, in the *middle*), resources along four links are allocated in the substrate network (*right*). In the right graph, solid lines represent virtual links mapped on single substrate links, solid curves are virtual links mapped on multiple substrate links, dotted lines are substrate links implementing a multihop virtual link, and dashed lines are unused substrate links.

Contribution. This paper initiates the study of a new problem, the discovery of a substrate topology through repeated VNet embedding requests. The considered problem is motivated by the emergence of network virtualization architectures which enables the flexible placement of virtual networks.

Our main contributions are :

(a) a new model and problem definition
(b) upper and lower bounds for trees, cactus graphs and general graphs
(c) a motif-based framework for further graph classes

(a) We introduce a formal model to study the problem of VNet topology inference. This model allows a customer or attacker to issue the following types of requests:

```
Is graph G (the VNet) embeddable
in graph H (the substrate)?
```

To measure how quickly a topology can be disclosed, we define the notion of *request complexity*: the number of VNet requests needed to disclose a substrate topology in the worst-case. Our model differs from existing topology discovery problems (such as, e.g., *graph tomography*, see Section 6) in that the requests come in the form of entire *graphs* instead of paths only.

An important assumption used in our work is the honesty of the provider, i.e., we assume that the provider will always accept a VNet request if it has sufficient resources for the embedding.

We show that while the VNet embedding relation differs from the *graph minor* relation, we can profit from its *poset* (partially ordered set) properties, and present algorithms achieving an optimal request complexity for different settings and substrate topologies.

(b) We derive request complexity bounds for different graph classes. In particular, we show that the request complexity of trees and cactus graphs is $O(n)$, while arbitrary graphs have a request complexity of $O(n^2)$, where n is the number of nodes in the substrate. In addition we provide matching lower bounds for these complexities.

(c) Moreover, we present a generalized graph inference framework based on the concept of so-called *motifs* that describe a graph based on components that are glued together.

In general, we understand our work as a first step towards a better understanding of the security aspects of VNet embeddings, and believe that our work opens interesting directions for future research.

Organization. The remainder of this paper is organized as follows. Section 2 introduces the VNet embedding problem, defines the request complexity, and derives some useful properties of the embedding relation. We present asymptotically optimal topology inference algorithms for trees and arbitrary graphs in Section 3. Our generalized motif framework appears in Section 4, together with an extended example for cactus graphs. We report on our simulation results in Section 5. After reviewing related work in Section 6, we conclude our work in Section 7.

2 Model

We first introduce the VNet topology discovery problem and subsequently describe our algorithmic approach.

2.1 VNet Embedding

Our formal setting consists of two entities: a *customer* (the "adversary") that issues virtual network (VNet) requests and a *provider* (a virtual network or infrastructure provider [27]) that performs the access control and the embedding of VNets. We model the virtual network requests as simple, undirected graphs $G = (V, E, w)$ (the so-called *guest graphs*) where V denotes the virtual nodes, E denotes the virtual edges connecting nodes in V, and w denotes the resource requirements of the node resp. link: the weight $w(v)$ describes the *demand* of node $v \in V$ for, e.g., computation or storage, and $w(e)$ describes the bandwidth demand of edge $e \in E$. Similarly, the infrastructure network is given as an undirected graph $H = (V, E, w)$ (the so-called *host graph* or *substrate*) as well, where V denotes the set of substrate nodes and E is the set of substrate links. Again, a weight function w can describe the available resources on a given node or edge.

Without loss of generality, we assume that neither the VNet nor the substrate topologies contain parallel edges or self-loops, and that H is connected. (As we will see, if H consists of multiple connected components, these components can be processed sequentially with our algorithms by keeping the VNet embedding in an already discovered component in order to focus on the next component.) Moreover, in this paper, in order to focus on the topological aspects, we will typically consider host graphs with unit capacities only, i.e., we will assume that $w \equiv 1$. As we will see, for substrate networks with unit capacities, we can also make the guest graph unweighted: all graph classes considered in this paper can be inferred with an asymptotically optimal complexity using unweighted VNets only.

In this paper we assume that besides the resource demands, the VNet requests do not impose any mapping restrictions, i.e., a virtual node can be mapped to *any* substrate node, and we assume that a virtual link connecting two substrate nodes can be mapped to an entire (but single) *path* on the substrate as long as the demanded capacity is available. These assumptions are common for virtual networks. [8]

A virtual link which is mapped to more than one substrate link however can entail certain costs at the *relay nodes*, the substrate nodes which do not constitute endpoints of the virtual link and merely serve for forwarding. For example, this cost may represent a header lookup cost and may be a function of the packet rate. However, depending on the application, the cost can also be more complex, e.g., in case of a VNet which requires additional functionality at the backbone routers, e.g., to implement an intrusion detection system. We model these kinds of costs with a parameter $\epsilon > 0$ (per link). Moreover, we also allow multiple virtual nodes to be mapped to the same substrate node if the node capacity allows it; we assume that if two virtual nodes are mapped to the same substrate node, the cost of a virtual link between them is zero.

Armed with these definitions, we can formalize our VNet embedding problem.

Definition 1 (Embedding π, Validity Properties, Embedding Relation $\mapsto$) An *embedding* of a graph $A = (V_A, E_A, w_A)$ to a graph $B = (V_B, E_B, w_B)$ is a mapping $\pi : A \to B$ where every node of A is mapped to exactly one node of B, and every edge of A is mapped to a path of B. That is, π consists of a node mapping $\pi_V : V_A \to V_B$ and an edge mapping $\pi_E : E_A \to P_B$, where P_B denotes the set of paths. We will refer to the set of virtual nodes embedded on a node $v_B \in V_B$ by $\pi_V^{-1}(v_B)$; similarly, $\pi_E^{-1}(e_B)$ describes the set of virtual links passing through $e_B \in E_B$ and $\pi_E^{-1}(v_B)$ describes the virtual links passing through $v_B \in V_B$ with v_B serving as a relay node.

To be *valid*, the embedding π has to fulfill the following four properties: (i) Each node $v_A \in V_A$ is mapped to exactly one node $v_B \in V_B$ (but given sufficient capacities, v_B can host multiple nodes from V_A). (ii) Links are mapped consistently, i.e., for two nodes $v_A, v'_A \in V_A$, if $e_A = \{v_A, v'_A\} \in E_A$ then e_A is mapped to a single (possibly empty and undirected) path in B connecting nodes $\pi(v_A)$ and $\pi(v'_A)$. A link e_A cannot be split into multiple paths. (iii) The capacities of substrate nodes are not exceeded: for all $v_B \in V_B$: $\sum_{u \in \pi_V^{-1}(v_B)} w(u) + \epsilon \cdot |\pi_E^{-1}(v_B)| \leq w(v_B)$. (iv) The capacities in E_B are respected as well, i.e., for all $e_B \in E_B$: $\sum_{e \in \pi_E^{-1}(e_B)} w(e) \leq w(e_B)$.

If there exists such a valid embedding π, we say that graph A can be embedded in B, indicated by $A \mapsto B$. Hence, $\mapsto$ denotes the VNet *embedding relation*.

Definition 2 (Embedding Cost) The cost associated with an embedding π is denoted by

$$\text{Cost}(\pi) = \sum_{v_A \in V_A} w(v_A) + \sum_{e_A \in E_A} w(e_A) \cdot |\pi^{-1}(e_A)| + \epsilon \cdot \sum_{v_B \in V_B} |\pi_E^{-1}(v_B)|$$

The provider has a flexible choice where to embed a VNet as long as a valid mapping is chosen. In order to design topology discovery algorithms, we can exploit the *poset* property of the embedding relation. The proof of the following lemma appears in the appendix.

Lemma 1 *The embedding relation $\mapsto$ applied to any graph family $\mathcal{G}$ (short: $(\mathcal{G}, \mapsto)$), forms a partially ordered set (a* poset*), i.e., it satisfies reflexivity, antisymmetry and transitivity.*

Observe that even though the VNet embedding is similar to the minor graph relation, there are some important differences. The following lemma shows that the minor graph relation and the embedding relation do not imply each other. The proof appears in the appendix.

Lemma 2 *Given two graphs $A,\ B \in \mathcal{G}$ with unit capacity it holds that (i) $A \mapsto B$ implies that A is a minor of B for $\epsilon > 0.5$. (ii) For smaller ϵ it holds that A may be embeddable in B even if A is not a minor of B. (iii) Not every minor A of a graph B can be embedded in A.*

2.2 Request Complexity

We assume the perspective of a customer (an "adversary") that seeks to disclose the (fixed) infrastructure topology of a provider with a minimal number of requests. These requests (and the answers to them) are the only means of obtaining information. As a performance measure, we introduce the notion of *request complexity*, i.e., the number of VNet requests which have to be issued until a given network is fully discovered, i.e., all vertices, edges and capacities are known to the adversary. The motivation for focusing on a request measure is based on the fact that a request constitutes the natural unit of negotiation between customer and provider, and issuing a request entails a certain overhead for both sides. In particular a situation where after the provider's answers, the customer repeatedly revokes her request, can be perceived as a denial-of-service attack. Algorithms achieving a low request complexity hence also ensure that adversarial information discovery activities remain hidden.

We are interested in algorithms that "guess" the target topology H (the host graph) among the set $\mathcal{H}$ of possible substrate topologies allowed by the model. Concretely, we assume that given a VNet request G (a guest graph), the substrate provider always responds with *an honest (binary) reply* R informing the customer whether the requested VNet G is embeddedable on the substrate H. In the following, we will use the notation

$$\texttt{request}(G, H)$$

to denote such an embedding request of G to H, and the provider will answer with the binary information whether G is embeddable in H (short: $G \mapsto H$). Based on this reply, the customer may then decide to ask the provider to embed the corresponding VNet G on H, or it may not embed it and continue asking for other VNets.

Let ALG be an algorithm that issues a series of request requests $G_1, \ldots, G_t$ each consisting of a request graph to reveal H.

Definition 3 (Request Complexity) The *request complexity* to infer the topology is measured in the number of requests t (in the worst case) until ALG issues a request G_t which is isomorphic to H and *terminates* (i.e., ALG knows that $H = G_t$ and does not issue any further requests).

2.3 Additional Complexities

An attacker only receives *one bit* of information: whether the VNet can be embedded or not. Thus, minimal information about the provider network is revealed, entailing a high request complexity. Alternative models may reduce the number of requests needed to infer the topology. For example, a plausible alternative may be a "valued reply model" where the provider returns the embedding *cost* $\text{Cost}(\texttt{request}(G, H))$. (The binary reply model can be emulated with this model by using the variable $(\text{Cost}(\texttt{request}(G, H)) < \infty)$.)

Moreover, we rely on the assumption that the provider always gives an *honest answer*. While there are algorithms to compute such honest answers [26], the problem is related to graph isomorphism testing and is generally NP-hard (see e.g., [3]). Thus, in practice, a provider may use approximation algorithms and heuristics to embed VNets (see e.g. [8]), and may not accept a VNet although it is theoretically embeddable. In addition, a provider being aware of potential attacks may proactively randomize its responses.

In other words, by focussing on the idealized case where providers answer embedding requests correctly and honestly, we investigate a worst case situation for the provider. Hence the results of this article determine how quickly and how much information on the host network can be revealed in circumstances that are ideal for attackers. These findings can help providers to decide on the tradeoff of a comprehensive client offering and the cost/benefit of attack countermeasures.

Besides the *request complexity* and the *embedding complexity*, there is a third relevant complexity which has not been discussed so far: the complexity of generating the attack sequence, i.e., generating the request G_{i+1} out of G_i given the provider answer. We will refer to this complexity as the *generation complexity*. While the generation complexity plays a secondary role in our paper, all algorithms presented in this paper are essentially greedy and efficient; an exception is the dictionary framework which relies on a pre-computed directed acyclic graph.

2.4 Terminology and Notation

The algorithms presented in this paper are often based on simple graphs (so-called *motifs*) such as a *chain*, a virtual link connecting two virtual nodes, or a *cycle*, three virtual nodes connected as a triangle.

Definition 4 (Chain C, Cycle Y, Diamond D, Bipartite Motif $K_{x,y}$, Cliques K_x) The motif C denotes a *C*hain graph, a virtual edge that maps to a path on the substrate: $C = (V = \{u, v\}, E = \{\{u, v\}\})$. Motif Y is a c*Y*cle, i.e., a virtual triangle that maps to a cycle in the substrate: $Y = (V = \{u, v, w\}, E = \{\{u, v\}, \{u, w\}, \{w, v\}\})$. The *D*iamond motif D is a complete graph over four nodes with one missing link. Finally, $K_{x,y}$ is a complete bipartite graph with $x + y$ nodes, and K_x is a clique with x nodes.

These basic motifs are then attached to each other to grow more complex VNet requests. We will often make use of a *graph grammars* notation [6] in order to describe the graph exploration iteratively.

When two motifs are attached to each other we use the term *concatenation* to describe that two motifs are glued to each other selecting one of the nodes of each of them as *attachment points* and merging them.

Definition 5 (Concatenation, $prefix$, $postfix$) We will use the notation M^j to denote the *concatenation* of j motifs M (where the attachment points must be clear from the context). Moreover, given two graphs G_1 and G_2 containing nodes u, v, the notation $G_1 v G_2(u)$ denotes a graph where the node $v \in G_1$ is merged with the node $u \in G_2$, i.e., the edges of G_2 are added to the set of G_1's edges and the corresponding nodes to the set of G_1's nodes; the node $u \in G_1$ is optional if clear from the context. Given a sequence S of motifs attached to each other, and a particular motif M belonging to this sequence, then $prefix(M, S)$ and $postfix(M, S)$ denote the subsequences of S before and after this motif M.

3 Adversarial Topology Discovery

This section presents tight bounds for the discovery of *trees* and *general graphs*.

3.1 Trees

We start with a simple observation: it is easy to decide whether the host graph forms a tree or not. A topology discovery algorithm ALG can test if the substrate $H \in \mathcal{H}$ is tree-like using a single request: ALG simply asks for a triangle network (i.e., a complete graph K_3 consisting of three virtual nodes, with unit virtual node and link capacities). The triangle can be embedded if and only if H contains a cycle. Once it is known that the set of possible infrastructure topologies (or host graphs) $\mathcal{H}$ is restricted to trees, the algorithm described in this section can be used to discover them. Moreover, as we will see, the algorithm presented in this section has the property that if $H \in \mathcal{H}$ does contain cycles, it automatically computes a *spanning tree* of H.

The tree discovery algorithm TREE (see Algorithm 1 for the formal listing) described in the following is based on the idea of incrementally growing the request graph by adding *longest chains* (i.e., "branches" of the tree). Intuitively, such a longest chain of virtual nodes will serve as an "anchor" for extending further branches in future requests: since the chain is maximal and no more nodes can be embedded, the number of virtual nodes along the chain must equal the number of substrate nodes on the corresponding substrate path. The endpoints of the chain thus cannot have any additional neighbors and must be tree leafs (we will call these nodes *explored*), and we can recursively explore the longest branches of the so-called *pending nodes* discovered along the chain.

More concretely, TREE first discovers the overall longest (cycle-free) chain of nodes in the substrate tree by performing binary search on the length of the maximal embeddable path. This is achieved by requesting, in request R_i, a VNet of 2^i linearly connected virtual nodes (of unit node and link capacities); in Algorithm 1, we refer to a single virtual link connecting two virtual nodes by a *chain* C, and a sequence of j chains by C^j. The first time a path of the double length 2^i is not embeddable, TREE asks for the longest embeddable chain with 2^{i-1} to $2^i - 1$ virtual nodes; and so on. Once the longest chain is found, its end nodes are considered *explored* (they cannot have any additional neighbors due to the longest chain property), and all remaining virtual nodes along the longest chain are considered *pending* (set $\mathcal{P}$): their tree branches still need to be explored. TREE then picks an arbitrary pending

node v and seeks to attach a maximal chain ("branch") analogously to the procedure above, except for that the node at the chain's origin is left pending until no more branches can be added. The scheme is repeated recursively until there are no pending nodes left. Formally, in Algorithm 1, we write GvC to denote that a chain C is added to an already discovered graph G at the virtual node v.

Algorithm 1 Tree Discovery: TREE

```
1: G := ({v}, ∅)   /* current request graph */
2: P := {v}   /* pending set of unexplored nodes*/
3: while P ≠ ∅ do
4:     choose v ∈ P, S :=exploreSequence(v)
5:     if S ≠ ∅ then
6:         G := GvS, add all nodes of S to P
7:     else
8:         remove v from P
```

exploreSequence(v)

```
1: S := ∅
2: if request(GvC, H) then
3:     find max j s.t. GvC^j ↦ H (binary search)
4:     S := C^j
5: return S
```

Theorem 1 *Algorithm* TREE *is correct and has a request complexity of $O(n)$, where n is the number of substrate nodes. This is asymptotically optimal in the sense that any algorithm needs $\Omega(n)$ requests in the worst case.*

Proof Correctness: Since the substrate network is connected, each node can be reached by a path from any other node. As the algorithm explores each path attached to a discovered node until no more nodes can be added, every node is eventually found. Since a tree is cycle-free, this also implies that the set of discovered edges is complete.

Complexity (upper bound): We observe that our algorithm has the property that at time t, it always ask for a VNet which is a strict super graph of any embeddable graph asked at time $t' < t$ (positive answer from the provider). Moreover, due to the exponential binary search construction, TREE issues $O(\log \ell)$ requests to discover a chain consisting of ℓ links. The cost of exploring a path can be distributed among its constituting links, thus we have an accounting scheme which shows that the amortized cost per link is constant: As there are at most $n-1$ links in a tree, the total number of requests due to the link discovery is linear in n as well. In order to account for requests at nodes that do not have any unexplored neighbors and lead to marking a node explored (at most one request per node), $O(n)$ requests need to be added.

Complexity (lower bound): The lower bound follows from the cardinality of the set of non-isomorphic trees, which is in the order of $2.96^n/n^{5/2}$ [24]. Since any discovery algorithm can only obtain a binary information for each request issued, a request cuts the remaining search space in (at most) half. Therefore, the request complexity of any algorithm is at least $\Omega(\log(2.96^n/n^{5/2})) = \Omega(n)$. □

Although TREE looks simple, one has to be careful to avoid pitfalls when trying to design tree discovery algorithms. For instance, consider a scheme where instead of searching for longest chains, we want to find the nodes of largest degree, e.g., by performing binary search on the neighborhood size of a given virtual node, and then recursively explore the discovered pending nodes by adding newly found nodes to the pending set: Since pending nodes are connected by virtual links to each other, they may physically be separated by many hops, and one has to ensure that the substrate nodes along these paths are not forgotten. Although these problems can be handled within the same asymptotical complexity, we do not elaborate on these directions more here.

Observe that TREE has the nice property that if H is not a tree, TREE simply computes a spanning tree of H by extending maximal (cycle-free) branches from the nodes.

Corollary 1 TREE *determines a spanning tree of any graph with request complexity $O(n)$.*

3.2 Arbitrary Graphs

Let us now turn to the general problem of inferring arbitrary substrate topologies. First note that even if the total number of substrate nodes is known, the adversary cannot simply compute the substrate edges by testing each virtual link between the node pairs: the fact that the corresponding virtual link can be embedded does not imply that a corresponding substrate link exists, because the virtual link might be mapped across an entire substrate *path*. Nevertheless, we will show in the following that a request complexity of $O(n^2)$ can be achieved; this is asymptotically optimal.

The main idea of our algorithm GEN is to build upon the TREE algorithm to first find a spanning tree (see Corollary 1). This spanning tree (consisting of pending nodes only) "reserves" the resources on the substrate nodes, such that they cannot serve as relay nodes for virtual links passing through them. Subsequently, we try to extend the spanning tree with additional edges. An arbitrary pending node u is chosen, and we try to add an edge to any other pending node

v in the spanning tree. After looping over all pending nodes and adding the corresponding links, u is marked *explored*. GEN terminates when no more pending nodes are left.

Theorem 2 *A general graph can be discovered with request complexity $O(n^2)$. This is asymptotically optimal.*

Proof Upper bound: According to Corollary 1, computing a spanning tree G using TREE requires $O(n)$ requests. Subsequently, GEN asks for each pair of pending nodes if an edge between them can be added without destroying embeddability. Since the spanning tree is used as a basis, and since $\epsilon > 0$, two nodes can be connected if and only if there is a direct substrate link between them; otherwise, the capacity constraints would be violated. Thus one request per possible substrate link is sufficient, which results in at most $O(n^2)$ requests.

Lower bound: The number a_n of non-isomorphic simple graphs on n vertices satisfies $2^{\binom{n}{2}}/n! \leq a_n \leq 2^{\binom{n}{2}}$. By the same argument as for the lower bound of the request complexity for trees, $\Omega(\log a_n) = \Omega(n^2)$ requests are necessary in the worst case. □

3.3 Discussion

We have shown that tree graphs can be discovered in time $O(n)$, while general graphs require time $O(n^2)$. This raises the question whether there exist additional graph families, which still allow for a sub-quadratic complexity. In the next section, we will answer this question affirmatively, by presenting a general discovery framework. As a corollary, we will show that for example, also the important class of cactus graphs can be explored in time $O(n)$.

We conclude with a discussion of the other two important complexities besides the request complexity: the *embedding complexity* and the *generation complexity*. Generally, the VNet embedding problem is NP-hard even for host graphs forming a simple line (i.e., a chain of physical nodes): the problem is equivalent to the *Minimum Linear Arrangement* problem (see e.g., [3]). Although some polynomial time solutions exist for special VNet topologies (e.g., note that our algorithms to discover trees rely on cycle-free VNets only for which there are good or even optimal algorithms [10]), for more complicated graphs such as cactus graphs, optimized Integer Programs with good (i.e., tight) LP relaxations may be required. Since our focus is on the attacker and not on the VNet provider, we do not study such formulations in more detail here but refer the reader to the corresponding literature (e.g., [26]).

More important for the efficient graph exploration is the generation complexity: how quickly can an adversary generate the next request? The algorithms TREE and GEN are essentially greedy and hence fast: an additional edge is added to a "pending" node which has not been fully explored yet. This can be done in time roughly linear with respect to the current VNet size (or faster if the pending components are organized efficiently in the data structure).

4 Motif Framework

We now present a framework for arbitrary graph inference. The framework is based on three principles: (1) growing request graphs iteratively, (2) using simple motifs to reserve entire subgraphs while exploring the detailed structure of the subgraph only later, (3) requesting more highly connected motifs first.

Concretely, our algorithm DICT is based on the observation that given a spanning tree, it is very costly to discover additional edges between nodes in 2-connected components: essentially, finding a single such edge requires testing all possible node pair combinations, which is quadratic in the component size. Thus, our algorithm iteratively first explores the basic "knitting" of the topology, i.e., it discovers first a sequence of maximal minors which are at least 2-connected (the *motifs*). DICT maintains the invariant that there are never two nodes u, v which are not k-connected in the currently requested graph H' while they are k-connected in H; no path relevant for the connectivity of the current component is overlooked and needs to be found later. Subsequently, nodes and edges which are not contributing to the connectivity of the current component can then be discovered by (1) *edge expansion* where additional nodes of degree two are added along a motif edge, and by (2) adding sequences of motifs to the nodes iteratively with the same process.

Figure 2 gives a simplified overview of our discovery strategy: in a first phase, we seek to discover the highly connected parts of the physical network (and reserve the corresponding resources, see Lemma 4); this is achieved by trying to embed sequences of so-called "motifs" of the highest connectivity and subsequently shifting toward less connected motifs. Nodes of degree two are found using *edge expansion*.

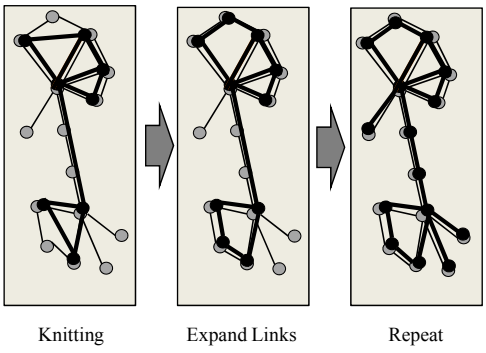

Fig. 2 Algorithm DICT first discovers and reserves sequences of highly connected parts of the host graph. Then, less connected parts are discovered by edge expansion and then the process is repeated until no additional nodes and edges can be found.

4.1 Overview

With the above intuition in mind, we give a short overview of this section. In Section 4.2, we first formally introduce the concept of *motifs*: simple graphs describing the possible knitting patterns of a host graph. For example, for a tree graph we only need a single motif, the *chain* C (two nodes connected by a link); since there are no 2-connected motifs in a tree, it can be discovered by attaching chains C to each other.

Another interesting graph family are *cactus graphs*: connected graphs in which any two simple cycles have at most one vertex in common. Cactus graphs are interesting in the sense that many backbone or core networks are reminiscent of cacti (see, e.g., the topologies collected in the Rocketfuel project [28]). In terms of motifs, since a cactus graph may contain cycles, besides the chain C we also need a cycle motif Y: three nodes connected in a triangle manner; however, C and Y are also sufficient to discover all cactus graphs.

More complicated graphs which include higher connected parts may require, e.g., motifs describing diamonds D or cliques K_4. We describe how to extract motifs for a given graph family in Lemma 3.

The first idea behind our algorithm DICT is that once a motif is embedded, it occupies resources on the entire host subgraph (Lemma 4 and Corollary 2), and the discovery of the remaining parts of the subgraph where the motif is embedded can be postponed to some later point: For example, Figure 2 (*left*) shows a motif occupying resources on the subgraph at the top; additional nodes (Figure 2 (*middle*)) and edges (Figure 2 (*right*)) can be discovered later.

However, in order to discover entire graphs, multiple motifs are needed: motifs must be attached to each other; accordingly, we introduce the concept of *motif sequences* and *attachment points* (Definition 8). Moreover, for DICT to be correct, it is important that more highly connected motifs are embedded first: a less connected motif may be embeddable on a subgraph but may "overlook" (i.e., not allocate resources on) some links. The situation is complicated by the fact that a graph A may not be embeddable on graph B and vice versa, but A may be embeddable on *two* graphs B which are attached to each other (see Figure 3). This motivates the dictionary concept (Section 4.3) where motifs are organized in an acyclic directed graph.

Our main result is an algorithm that correctly discovers a given topology by respect the order of the dictionary of the corresponding graph family(Theorem 3).

We will give a concrete examples for our concepts and our approach throughout this section and describe the explicit requests issued by our algorithm for a given network in Section 4.6.

4.2 Motifs: Composition and Expansion

We now start to make the intuition provided above more formal.

Definition 6 (Motif) Given a graph family $\mathcal{H}$, the set of motifs of $\mathcal{H}$ is defined constructively: If any member of $H \in \mathcal{H}$ has an edge cut of size one, the *chain* C is a motif for $\mathcal{H}$. All remaining motifs are at least 2-connected (i.e., any pair of nodes in a motif is connected by at least two vertex-disjoint paths). These motifs can be derived by the at least 2-connected components of any $H \in \mathcal{H}$ by repeatedly removing all nodes with degree smaller or equal than two from H and merging the incident edges, as long as all remaining cycles do not contain parallel edges. Only one instance of isomorphic motifs is kept.

In other words, a graph family containing all elements of $\mathcal{H}$ can be constructed by applying the following rules repeatedly.

Definition 7 (Rules) (1) Create a new graph consisting of a motif $M \in \mathcal{M}$ (*New Motif Rule*). (2) Given a graph created by these rules, replace an edge e of H by a new node and two new edges connecting the incident nodes of e to the new node (*Insert Node Rule*). (3) Given two graphs created by these rules, attach them to each other such that they share exactly one node (*Merge Rule*).

These rules are sufficient to compose all graphs in $\mathcal{H}$: If $\mathcal{M}$ includes all motifs of $\mathcal{H}$, it also includes all 2-connected components of H, according to Definition 6. These motifs can be glued together using the

Merge Rule, and eventually the low-degree nodes can be added using the *Insert Node Rule*. Therefore, we have the following lemma.

Lemma 3 *Given the motifs $\mathcal{M}$ of a graph family $\mathcal{H}$, the repeated application of the rules in Definition 7 allows us to construct each member $H \in \mathcal{H}$.*

The following lemma shows that when degree-two nodes are added to a motif M to form a graph G, all network elements (substrate nodes and links in the graph defining the motif) are *used* when embedding M in G (i.e., $M \mapsto G$).

Lemma 4 *Let $M \in (\mathcal{M} \setminus \{C\})$ be an arbitrary two-connected motif, and let G be a graph obtained by applying the* Insert Node Rule *(Rule 2 of Definition 7) to motif M. Then, an embedding $M \mapsto G$ involves all nodes and edges in G: at least ϵ resources are used on all nodes and edges.*

Lemma 4 implies that no additional nodes can be inserted to an existing embedding. In other words, a motif constitutes a "minimal reservation pattern". As we will see, our algorithm will exploit this invariant that motifs cover the entire graph knitting, and adds simple nodes (of degree 2) only in a later phase.

Corollary 2 *Let $M \in (\mathcal{M} \setminus \{C\})$ and let G be a graph obtained by applying Rule 2 of Definition 7 to motif M. Then, no additional node or motif can be embedded on G after embedding $M \mapsto G$.*

Next, we want to *combine* motifs to explore larger "knittings" of graphs. Each motif pair is glued together at a single node *or edge* ("attachment point"): We need to be able to conceptually join to motifs at edges as well because the corresponding edge of the motif can be expanded by the *Insert Node Rule* to create a node where the motifs can be joined.

Definition 8 (Motif Sequences, Attachment Points, Subsequences) A *motif sequence* S is a list $S = (M_1 a_1 a'_1 M_2 \dots M_k)$ where for all i: $M_i \in \mathcal{M}$, and where M_i is glued together at exactly one node with M_{i-1} (i.e., M_i is "attached" to a node of motif M_{i-1}): the notation $M_{i-1} a_{i-1} a'_{i-1} M_i$ specifies the selected attachment points a_{i-1} and a'_{i-1}. If the attachment points are irrelevant, we use the notation $S = (M_1 M_2 \dots M_k)$ and M_i^k denotes an arbitrary sequence consisting of k instances of M_i. If S can be decomposed into $S = S_1 S_2 S_3$, where S_1, S_2 and S_3 are (possibly empty) motif sequences as well, then S_1, S_2 and S_3 are called *subsequences* of S. This relationship is denoted by writing, e.g., $S_1 \prec S$.

In the following, we will sometimes use the *Kleene star* notation $X^\star$ to denote a sequence of (zero or more) elements of X attached to each other.

One has to be careful when arguing about the embedding of motif sequences, as illustrated in Figure 3 which shows a counter example for $M_i \not\mapsto M_j \Rightarrow M_i \not\mapsto M_j^k$ for any k. More precisely, there exist motifs that cannot be embedded into single instances of each other but at least one of them can be embedded in sequences of the other motif. Consider the problem instance where we have two motifs A and B where $A \not\mapsto B$, but $A \mapsto B^2$ and the host network is B^2. If we proceed purely incrementally, then we might not fully discover the host network, depending on the order of embedding requests. If we request A first, we will get a positive reply and cannot embed any other motifs subsequently. Only if we request B first, we will be able to discover B^2 with this approach. Thus an algorithm that adds motifs to its request networks purely incrementally in an arbitrary order cannot discover all substructures. This is the motivation for introducing the concept of a *dictionary* which imposes an order on motif sequences and their attachment points.

4.3 Dictionary Structure and Existence

In a nutshell, a dictionary is a *Directed Acyclic Graph (DAG)* defined over all possible motifs $\mathcal{M}$ and imposes an order (poset relationship $\mapsto$) on problematic motif sequences which need to be embedded one before the other (e.g., the composition depicted in Figure 3). To distinguish them from sequences, dictionary entries are called *words*.

Definition 9 (Dictionary, Words) A *dictionary* $D(V_D, E_D)$ is a *Directed Acyclic Graph (DAG)* over a set of motif sequences V_D together with their attachment points. In the context of the dictionary, we will call a motif sequence *word*. The links E_D represent the poset embedding relationship $\mapsto$.

Concretely, the DAG has a single root r, namely the chain graph C (with two attachment points). In general, the *attachment points* of each vertex $v \in V_D$ describing a word w, define how w can be connected to other words. The directed edges $E_D = (v_1, v_2)$ represent the transitively reduced embedding poset relation with the chain C context: Cv_1C is embeddable in Cv_2C and there is no other word Cv_3C such that $Cv_1C \mapsto Cv_3C$, $Cv_3C \mapsto Cv_2C$ and $Cv_3C \not\mapsto Cv_1C$ holds. (The chains before and after the words are added to ensure that attachment points are "used": there is no edge between two isomorphic words with different attachment point pairs.)

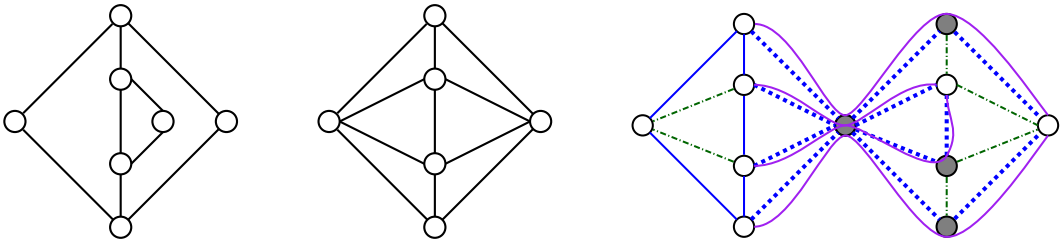

Fig. 3 *Left:* Motif A. *Center:* Motif B. Observe that $A \not\to B$. *Right:* Motif A is embedded into two consecutive Motifs B: solid lines are virtual links mapped on single substrate links, s olid curves are virtual links mapped on multiple substrate links, dotted lines are substrate links implementing a multi-hop virtual link, and dashed lines are substrate unused links. Grayed nodes are relay-only nodes. Observe that the central node has a relaying load of 4ϵ.

We require that the dictionary be *robust to composition*: For any node v, let $R_v = \{v' \in V_D, v \mapsto v'\}$ denote the "reachable" set of words in the graph and $\overline{R}_v = V_D \setminus R_i$ all other words. We require that $v \not\to W$ for all $W \in Q_i := \overline{R}_i^\star \setminus R_i^\star$, where the transitive closure operator $X^\star$ denotes an arbitrary sequence (including the empty sequence) of elements in X (according to their attachment points).

See Figure 4 for an example. Informally, the robustness requirement means that the word represented by v cannot be embedded in any sequence of "smaller" words, unless a subsequence of this sequence is in the dictionary as well. As an example, consider a dictionary containing the motifs A and B from Figure 3: this dictionary would not only contain vertices A and B but also BB, as well as a path from A to BB. In the following, we use the notation $\max_{v \in V_D}(v \mapsto S)$ to denote the set of "maximal" vertices with respect to their embeddability into S: $i \in \max_{v \in V_D}(v \mapsto S) \Leftrightarrow (i \mapsto S) \wedge (\forall j \in \Gamma^+(i), j \not\to S)$, where $\Gamma^+(v)$ denotes the set of outgoing neighbors of v. Furthermore, we say that a dictionary D *covers* a motif sequence S iff S can be formed by concatenating dictionary words (henceforth denoted by $S \in D^\star$) at the specified attachment points. More generally, a dictionary covers a graph, if it can be formed by merging sequences of $D^\star$.

Let us now derive some properties of the dictionary which are crucial for a proper substrate topology discovery. First we consider maximal dictionary words which can serve as embedding "anchors" in our algorithm.

Lemma 5 *Let D be a dictionary covering a sequence S of motifs, and let $i \in \max_{v \in V_D}(v \mapsto S)$. Then i constitutes a subsequence of S, i.e., S can be decomposed to $S_1 i S_2$, and S contains no words of order at most i, i.e., $S_1, S_2 \in (\overline{R}_i \cup \{i\})^\star$.*

The following corollary is a direct consequence of the definition of $i \in \max_{v \in V_D}(v \mapsto S)$ and Lemma 5: for a motif sequence S with $S \in (\overline{R}_i \cup \{i\})^\star$, all the subsequences of S that contain no i are in $\overline{R}_i^\star$. As we will see, the corollary is useful to identify the motif words composing a graph sequence, from the most complex words to the least complex ones.

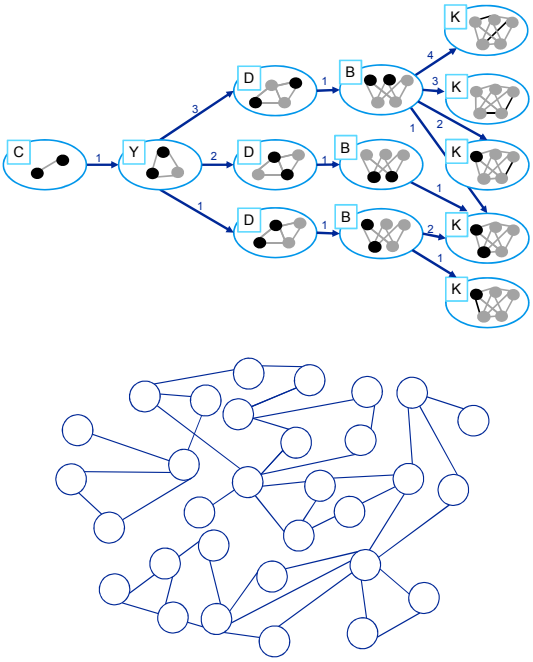

Fig. 4 a) Example dictionary with motifs Chain C, Cycle Y, Diamond D, complete bipartite graph $B = K_{2,3}$ and complete graph $K = K_5$. The attachment point pair of each word is black, the other nodes and edges of the words are grey. The edges of the dictionary are locally labeled, which is used in DICT later. b) A graph that can be constructed from the dictionary words.

Corollary 3 *Let D be a dictionary covering a motif sequence S, and let $i \in \max_{v \in V_D}(v \mapsto S)$. Then S can be decomposed as a sequence $S = T_1 i T_2 i, \ldots, i T_k$ with $T_j \in Q_i$ for all $j = 1, \ldots, k$.*

This corollary can be applied recursively to describe a motif sequence as a sequence of dictionary entries. Note that a dictionary always exists.

Lemma 6 *There exists a dictionary $D = (V_D, E_D)$ that covers all member graphs H of a motif graph family $\mathcal{H}$ with n vertices.*

Note that the proof of Lemma 6 only addresses the composition robustness for sequences of up to n nodes. However, it is clear that $|V(G)| > |V(H)| \Rightarrow G \not\hookrightarrow H$. Therefore it cannot happen that a request of a sequence with more than n nodes is answered positively on a host network of n nodes and hence this limitation does not pose an issue when devising dictionary-based algorithms for VNet topology discovery.

4.4 The Dictionary Algorithm

With these concepts in mind, we are ready to present DICT in detail (cf Algorithm 2). Basically, DICT always grows a request graph $G = H'$ until it is isomorphic to H (the graph to be discovered). This graph growing is performed according to the dictionary, i.e., we try to embed new motifs in the order imposed by the dictionary DAG.

Let us specify the *topological order* in which algorithm DICT discovers the dictionary words. First, for each node v in V_D, we define an order on its outgoing edges $\{(v,w)|w \in \Gamma^+(v)\}$. This order is sometimes referred to as a *port labeling*, and each path from the dictionary root (the chain C) to a node in V_D can be represented as the sequence of port labels at each traversed node $(l_1, l_2, \ldots, l_l)$, where l_1 corresponds to a port number in C. We can simply use the lexicographic order on integers, $<^d$: $(a_1, a_2, \ldots, a_{n_1}) <^d (b_1, b_2, \ldots, b_{n_2}) \iff ((\exists\, m > 0)\ (\forall\, i < m)(a_i = b_i) \wedge (a_m < b_m)) \vee (\forall i \in \{1, \ldots n_1\}, (a_i = b_i) \wedge (n_1 < n_2))$, to associate each vertex with its minimal sequence, and sort vertices of V_D according to their embedding order. Let r be the *rank* function associating each vertex with its position in this sorting: $r : V_D \to \{1, \ldots |V_D|\}$ (i.e., r is the topological ordering of D).

The fact that subsequences can be defined recursively using a dictionary (Lemma 5 and Corollary 3) is exploited by algorithm DICT. Concretely, we apply Corollary 3 to gradually identify the words composing a graph sequence, from the most complex words to the least complex ones. This is achieved by traversing the dictionary depth-first, starting from the root C up to a maximal node: algorithm DICT tests the nodes of $\Gamma^+(v)$ in increasing port order as defined above. As a shorthand, the word $v \in V_D$ with $r(v) = i$ is written as $D[i]$; similarly $D[i] < D[j]$ holds if $r(D[i]) < r(D[j])$, a notation that is useful since $D[j]$ is detected before $D[i]$ by algorithm DICT. As a consequence, the first matched word of a sequence S is unique: it is $i = \arg\max_x(D[x] \mapsto S) = \max\{r(v)|v \in \max_{v' \in V_D}(v' \mapsto S)\}$, the maximal word in S.

Algorithm DICT distinguishes whether the subsequences next to a word $v \in V_D$ are empty ($\emptyset$) or chains (C), and we will refer to the subsequence before v by BF and to the subsequence after v by AF. Concretely, while recursively exploring a sequence between two already discovered parts $T_<$ and $T_>$ we check whether the maximal word v is directly next to $T_<$ (i.e., $T_< v, \ldots, T_>$) or $T_>$ or both ($\emptyset$), or whether v is somewhere in the middle. In the latter case, we add a chain (C) to be able to find the greatest possible word in a next step.

DICT uses tuples of the form $(i, j, \text{BF}, \text{AF})$ where i, j are integers and $(\text{BF}, \text{AF}) \in \{\emptyset, C\}^2$, i.e., $D[i]$ denotes the maximal word in D, j is the number of consecutive occurrences of the corresponding word, and BF and AF represent the words before and after $D[i]$. These tuples are lexicographically ordered by the total order relation $>$ on the set of possible $(i, j, \text{BF}, \text{AF})$ tuples defined as follows: let $t = (i, j, \text{BF}, \text{AF})$ and $t' = (i', j', \text{BF}', \text{AF}')$ two such tuples. Then $t > t'$ iff $w > w'$ or $w = w' \wedge j > j'$ or $w = w' \wedge j = j' \wedge \text{BF} = C \wedge \text{BF}' = \emptyset$ or $w = w' \wedge j = j' \wedge \text{BF} = \text{BF}' \wedge \text{AF} = C \wedge \text{AF}' = \emptyset$.

With these definition we can prove that algorithm DICT is correct.

Theorem 3 *Given a dictionary for $\mathcal{H}$, algorithm* DICT *correctly discovers any $H \in \mathcal{H}$.*

Algorithm 2 Motif Graph Discovery DICT

1: $H' := \{\{v\}, \emptyset\}$ /*current request graph*/,
   $\mathcal{P} := \{v\}$ /*set of unexplored nodes*/
2: **while** $\mathcal{P} \neq \emptyset$ **do**
3: choose $v \in \mathcal{P}$, $T :=$ *findMotifSequence*$(v, \emptyset, \emptyset)$
4: **if** $(T \neq \emptyset)$ **then** $H' := H'vT$, add all nodes of T to $\mathcal{P}$, **for all** $e \in T$ **do** *edgeExpansion*(e)
5: **else** remove v from $\mathcal{P}$

findMotifSequence$(v, T_<, T_>)$

1: find maximal $i, j, \text{BF}, \text{AF}$ s.t. $H'v\,(T_<)\ \text{BF}\ (D[i])^j\ \text{AF}\ (T_>) \mapsto H$ where $\text{BF}, \text{AF} \in \{\emptyset, C\}^2$ /* issue requests */
2: **if** $((i, j, \text{BF}, \text{AF}) = (0, 0, C, \emptyset))$ **then return** $T_< C T_>$
3: **if** $(\text{BF} = C)$ **then** $\text{BF} =$ *findMotifSequence*$(v, T_<, (D[i])^j\ \text{AF}\ T_>)$
4: **if** $(\text{AF} = C)$ **then** $\text{AF} =$ *findMotifSequence*$(v, T_<\ \text{BF}\ (D[i])^j, T_>)$
5: **return** $\text{BF}\ (D[i])^j\ \text{AF}$

edgeExpansion(e)

1: let u, v be the endpoints of edge e, remove e from H'
2: find maximal j s.t. $H'vC^j u \mapsto H$ /* issue requests */
3: $H' := H'vC^j u$, add newly discovered nodes to $\mathcal{P}$

4.5 Request Complexity

The focus of DICT is on generality rather than performance, and indeed, the resulting request complexities can often be high. However, as we will see, there are interesting graph classes which can be solved efficiently.

Let us start with a general complexity analysis. The requests issued by DICT are constructed in Line 1 of $finding\_motif\_sequence()$ and in Line 2 of *edgeExpansion*(). We will show that the request complexity of the latter is linear in the number of edges of the host graph while the request complexity of $finding\_motif\_sequence()$ depends on the structure of the dictionary. Essentially, an efficient implementation of Line 1 of $finding\_motif\_sequence$ in DICT can be seen as the depth-first exploration of the dictionary D starting from the chain C. More precisely, at a dictionary word v requests are issued to see if one of the outgoing neighbors of v could be embedded at the position of v. As soon as one of the replies is positive, we follow the corresponding edge and continue recursively from there, until no outgoing neighbors can be embedded. Thus, the number of requests issued before we reach a vertex v can be determined easily.

Recall that DICT tests vertices of a dictionary D according to a fixed port labeling scheme. For any $v \in V_D$, let $p(C, v)$ be the set of paths from C to v (each path including C and v). In the worst case, discovering v costs $cost(v) = \max_{p \in p(C,v)}(\sum_{u \in p} |\Gamma^+(u)|)$.

Lemma 7 *The request complexity of Line 1 of* findMotifSequence$(v', T_<, T_>)$ *to find the maximal* $i, j, \text{BF}, \text{AF}$ *such that* $H'v'$ $(T_<)$ BF $(D[i])^j$ AF $(T_>) \mapsto H$ *where* $\text{BF}, \text{AF} \in \{\emptyset, C\}^2$ *and* H' *is the current request graph is* $O(\max_{v \in V_D} cost(v) + j)$.

When additional nodes are discovered by a positive reply to an embedding request, then the request complexity between this and the last previous positive reply can be amortized among the newly discovered nodes. Let $num\_nodes(v)$ denote the number of nodes in the motif sequence of the node v in the dictionary.

Theorem 4 *The request complexity of algorithm* DICT *is at most* $O(n \cdot \Delta + m)$*, where* m *denotes the number of edges of the inferred graph* $H \in \mathcal{H}$*, and* Δ *is the maximal ratio between the cost of discovering a word* v *in* D *and* $num\_nodes(v)$*, i.e.,* $\Delta = \max_{v \in V_D}\{cost(v)/num\_nodes(v)\}$.

4.6 Examples for Theorems 3 and 4

Let us consider concrete examples to provide some intuition for Theorem 3 and Theorem 4. We will denote chains, cycles, diamonds, the complete bipartite graph over two times three nodes and the complete graph over five nodes by C, Y, D, B and K respectively. If we ignore the attachment points, then executing DICT on the graph in Figure 4.b), leads to the following request sequence for the first call of *findMotifSequence*(): $C \to Y \to D \to B \to K$ (negative reply) $\to BB$ (negative reply) $\to CKC$ (recursion in Line 3) $\to YKC$ (negative reply) $\to YCKC \to DCKC \to D^2CKC$ (negative reply) $\to CDCKC \to \ldots$. The motif sequence tree of this example is illustrated in Figure 5.

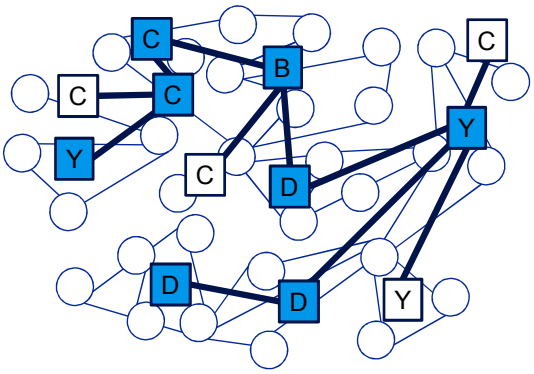

Fig. 5 Motif sequence tree of the graph in Figure 4 *b*). The squares and the edges between them depict the motif composition, the shaded squares belong to the motif sequence YC^2BDYD^2 discovered in the first execution of *findMotifSequence*() (chains, cycles, diamonds, and the complete bipartite graph over two times three nodes are denoted by C, Y, D and B respectively). Subsequently, the found edges are expanded before calling *findMotifSequence*() another four times to find Y and three times C.

A fundamental graph class are *trees*. Since, the tree does not contain any 2-connected structures, it can be described by a single motif: the chain C. Indeed, if DICT is executed with a dictionary consisting in the singleton motif set $\{C\}$, it is equivalent to a recursive version of TREE from [21] and seeks to compute maximal paths.

Corollary 4 *Trees can be described by one motif (the chain* C*). The request complexity of* DICT *on trees is* $O(n)$.

Proof Since there is only one motif and it has only one attachment point pair, Δ of the dictionary is constant. Consequently, a linear request complexity follows directly from Theorem 4 due to the planarity of cactus graphs (i.e., $m \in O(n)$). □

An example where the dictionary is efficient although the connectivity of the topology can be high, are *block graphs*. A block graph is an undirected graph

in which every bi-connected component (a *block*) is a clique. A *generalized block graph* is a block graph where the edges of the cliques can contain additional nodes. In other words, in the terminology of our framework, the motifs of generalized block graphs are *cliques*. For instance, cactus graphs are generalized block graphs where the maximal clique size is three.

Corollary 5 *Generalized block graphs can be described by the motif set of cliques. The request complexity of* DICT *on generalized block graphs is $O(m)$, where m denotes the number of edges in the host graph.*

Proof The framework dictionary for generalized block graphs consists of the set of cliques, as a clique with k nodes cannot be embedded on sequences of cliques with less than k nodes. As there are three attachment point pairs for each complete graph with four or more nodes, DICT can be applied using a dictionary that contains three entries for each motif with more than three nodes ($num\_nodes() > 3$). Thus, the i^{th} dictionary entry has $\lfloor i/3 \rfloor + 3$ nodes for $i > 1$ and $cost(D[i]) < 3(i+2)$ and Δ of D is hence in $O(1)$. Consequently the complexity for generalized block graphs is $O(m)$ due to Theorem 4. □

On the other hand, Theorem 4 also states that highly connected graphs may require $\Omega(n^2)$ requests, even if the dictionary is small.

4.7 Extended Example: Cactus Graphs

We conclude the discussion of the framework with an extended example for cactus graphs. As already mentioned, cactus graphs are a particularly interesting topology in the context of the Internet. For example, the topologies collected in experiments such as *Rocketfuel* are often sparse but contain certain cycles along the backbone, and thus resemble the cactus graph. [28] Formally, a cactus graph is a connected graph in which any two simple cycles have at most one vertex in common. Or equivalently: every edge in the cactus graph belongs to at most one 2-connected component, i.e., the cactus graph does not contain any diamond graph shaped minors.

The motif set for cactus contains only cycles (short form: Y) and chains (short form: C). Since cycles are symmetric, the dictionary required to discover cactus graphs is very simple: $\{C, Y\}$. Due to this simplicity it is possible to unfold the recursions in DICT to produce a iterative and compact algorithm to discover cactus topologies: the algorithm CACTUS. This might prove useful for implementations and provides a different perspective to the reader. Note that in contrast to a tree where nodes are origins of simple paths (i.e., branches), a cactus graphs is very simple: C, Y. Due to this simplicity it is possible to unfold the recursions in DICT to produce a standalone iterative and compact algorithm to discover cactus topologies: the CACTUS algorithm. Note that in contrast to a tree where nodes are origins of simple paths (i.e., branches), a cactus node can be the origin of several sub-cactus graphs consisting of 1- and 2-connected components. That is, the resulting graph when collapsing one or several 2-connected components to a single node is a cactus as well, or even a tree if all components are collapsed. The structure of CACTUS is therefore similar to TREE in its iterative approach. Concretely, instead of using longest chains as "anchor points" for extending an existing topology, we search, in each possible direction from a pending cactus node v for a maximal sequences of cycles and chains. Only once such a sequence (or "*motif*") is found, our algorithm CACTUS finds out the detailed structure of the chain/cycle sequence by inserting as many nodes on the chains and cycles as possible.

The formal listing of CACTUS appears in Algorithm 3. Since the algorithm is no longer recursive as the DICT algorithm and only requires two motifs, we can prove its correctness and request complexity more easily.

Theorem 5 CACTUS *discovers any cactus topology with request complexity $O(n)$. This is asymptotically optimal.*

Proof Correctness: Since by definition, the cactus graph does not contain any diamond graph shaped minor graphs, no two finite faces of the cactus graph can share a link (but sharing nodes is possible), and hence, any node v connects a set of (sub)cactus graphs.

Starting with one node, CACTUS repeatedly finds a sequence S of cycles (Y) and chains (C) with *exploreSequence()* and then expands these cycles and chains to find all nodes lying on them with *edgeExpansion*$()$. This algorithm terminates as soon as all edges incident to nodes found so far (i.e., *pending* nodes) have been discovered. Consequently, we need to show that all nodes and all their adjacent edges are detected in order to prove correctness (i.e., there is a bijection between the edges in G and in H and thus it is not possible that a virtual edge connects two nodes that are not adjacent in the (sub)cactus graphs).

Note that the algorithm maintains the invariant that $GvS \mapsto H$ at all times. As a consequence we can

Algorithm 3 Cactus Discovery: CACTUS

1: $G := \{\{v\}, \emptyset\}$ /* current request graph */
2: $\mathcal{P} := \{v\}$ /* pending set of unexplored nodes*/
3: **while** $\mathcal{P} \neq \emptyset$ **do**
4: choose $v \in \mathcal{P}$, $S :=$*exploreSequence*(v)
5: **if** $S \neq \emptyset$ **then**
6: $G := GvS$, add all nodes of S to $\mathcal{P}$
7: **for all** $e \in S$ **do** *edgeExpansion*(e)
8: **else**
9: remove v from $\mathcal{P}$

exploreSequence(v)

1: $S := \emptyset$, $P'' := \emptyset$
2: **if** $GvYCY \mapsto H$ **then**
3: find max j s.t. $GvY^jCY \mapsto H$
4: $S := Y^jCY$, $P' := \{C\}$
5: **while** $P' \neq \emptyset$ **do**
6: **for all** $C_i \in P'$ **do**
7: $A := prefix(C_i, S)$, $B := postfix(C_i, S)$;
8: **if** $GvACYCB \mapsto H$ **then**
9: find max j, k s.t. $GvAC(Y^jC)^kB \mapsto H$
10: **for** $l := 1, \ldots, k$ **do**
11: $P'' := P'' \cup \{C_l\}$
12: $S := AC(Y^jC)^kB$
13: $P' := P''$, $P'' := \emptyset$
14: **if** request$(GvSY, H)$ **then**
15: find max j s.t. $GvSY^j \mapsto H$
16: $S := SY^j$
17: **if** request$(GvSC, H)$ **then**
18: $S := SC$
19: **return** S

edgeExpansion(e)

1: let u, v be the endpoints of edge e, remove e from G
2: find max j s.t. $GvC^ju \mapsto H$
3: $G := GvC^ju$, add newly discovered nodes to $\mathcal{P}$

analyze the properties of S and thereby deduce properties of the substrate. We start by proving that for a sequence S discovered in *exploreSequence*(v) the following properties hold: (i) no more Ys can be inserted (replace a Y by a YY or a C by a CYC), (ii) no chain can be inserted between two cycles (replace YY with YCY) and (iii) no C can be replaced by a Y. These invariants show that the discovered sequence S cannot be extended with more cycles or chains between cycles. Based on these invariants it remains to show that in the following steps of the algorithm we discover all nodes which are part of this sequence.

(i) If there are cycles attached to v directly, their maximal undetected concatenated occurrence is discovered in Line 3 (i.e., if $GvY^j \mapsto H$, j maximum, it is not possible that the remaining topology of H after the embedding $\pi(G)$ contains a sequence starting at v with more than j concatenated cycles). Within one execution of the *forall loop* (Line 6-13) the maximal value j reaches, decreases in the next execution of the loop. For each chain C_i treated in the forall loop it holds that all Y^j occurrences are discovered (guaranteed by Line 9) and thus it is not possible to replace any Y by YY. Replacing a C by a CYC is not possible since this is checked for all C in Line 9 of *exploreSequence*(v). The last C that might be appended to S in Line 19 cannot be replaced by CYC as this would have happened in Line 9. (ii) Replacing $S = AYYB$ by $AYCYB$ is only possible if the substrate contains $AYYYB$ and B starts with C, i.e., it would invalidate (i) and is thus impossible. (iii) Every C of S is between two Y. If C could be replaced by a Y this would have been discovered in Line 3 or 9 and is thus not possible anymore at the end of *exploreSequence*(v).

As a consequence of Properties $(i), (ii), (iii)$ it holds that with S all nodes of degree one or nodes that lie at the intersection of cycles and chains or cycles and cycles (i.e., nodes with degree three or four in S) are detected in *exploreSequence*(v). For a given sequence S the remaining nodes of degree two in S are discovered in *edgeExpansion*(e). It thus remains to prove that among these discovered nodes the ones with higher degrees are further explored for additional sequences attached to them. As all nodes are added to $\mathcal{P}$ when they are discovered (Line 6 of Algorithm 3, and Line 3 of the subroutine *edgeExpansion*(e)), and as the while loop in Line 3 is repeated until there are no outgoing sequences from a node anymore, all cactus edges are detected. Any additional edge would lead to a diamond minor, resulting in a contradiction.

Complexity: We can again use an amortization scheme that assigns request costs to edges: an edge is assigned the cost of the request where it is identified for the first time (in *exploreSequence*(v) this can happen at Lines 2, 3, 8, 9, 15, 16, 18, in *edgeExpansion*(e) in Line 2). In order to find the maximum embeddable sequence, there is one request with a negative answer in the execution of Lines 3, 9, 16 in *exploreSequence*(v) and Line 2 in *edgeExpansion*(e). Let us assign these unsuccessful requests that do not discover a new edge to the last edge discovered before. Thus in the worst case, there are two requests for each edge. Since the number of edges in cactus graphs is linear in the number of nodes, a request complexity of $O(n)$ follows.

Optimality: Since cactus graphs constitute a superset of tree graphs, the lower bound of Theorem 1 still applies, i.e., $\Omega(n)$ requests are needed by any algorithm. □

5 Simulations

To complement our theoretical results and to validate our framework on real network topologies, we dissected the *ISP topologies* provided by the Rocketfuel [28] mapping engine. In particular, given the high

execution times of DICT in the worst-case and for complex motif sets, we were interested in (1) the structure and complexity of the motif sets of these topologies, and in (2) the question whether an efficient and approximate variant of DICT could be used in practice to infer large parts of the network in linear time.

For this purpose, we implemented our motif extraction algorithm. Concretely, for each Rocketfuel topology, we first isolate the subgraphs that are trees (i.e., the 1-connected subgraphs), by recursively removing leaf-nodes. The remainder of the topology is then simplified by contracting degree-2 nodes that would be detected during the algorithms' edge expansion phases. The result is a topology only containing motifs connected by "articulation vertices", which provide us with a mean to distinguish motifs. Finally, the extracted motifs are compared pairwise to distinguish between isomorphic and non-isomorphic ones. (The corresponding R script is made publicly available at `homepages.laas.fr/~gtredan/topoInference.R`.)

Figure 6 *a)* provides some statistics about the considered AS router-level topologies: their overall size, the number of nodes belonging to a 1-connected subgraph, the number of nodes belonging to a motif, and the number of nodes belonging to the largest motif of each topology. For instance, the figure shows that the AS 1221 topology contains 2669 nodes, but only 310 nodes are part of a bi-connected component. Among those nodes, 47 are expanded degree-2 nodes, and 48 are part of motifs containing only 3 or 4 nodes. The remaining 215 nodes compose the "network heart" (containing 716 edges out of the topologies' 3175 edges): such a single highly-connected motif is typical and appears for many AS topologies. One takeaway from this plot is that, since most Rocketfuel networks are built of simple motifs and since DICT discovers both tree and relay nodes easily, most of each topology could be discovered quickly with an *approximate* DICT algorithm containing merely the basic motifs: only the few more complex motifs in the network heart require an exhaustive link exploration. In other words, the vast majority of the topology can be discovered in linear time.

Figures 6 *b)* and *c)* further explore the fraction of nodes that can be discovered by DICT restricted to a small motif set. For this purpose, we built a dictionary containing all the motifs identified in our dataset, and removed the biggest motif of each topology (the "network heart"). The remaining 19 motifs are depicted Figure 6 *b)*: they are surprisingly simple and symmetric. Figure 6 *c)* shows the fraction of nodes in perfectly inferred subgraphs: we see that a 19-motifs dictionary is sufficient to explore from 37 to 92% of the nine considered AS topologies. Figure 6 d) represents the frequency distribution of the most common motifs in the nine topologies. Note that interestingly, motif Y only comes third, with 11 occurrences across the dataset.

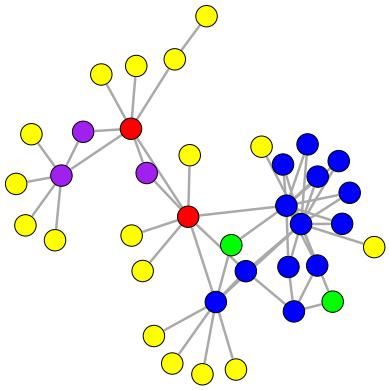

Fig. 7 Representation of the AS-4755 network where tree nodes are colored yellow, relay-nodes are green, attachment point nodes are red, cYcle motifs nodes are purple and the maximal motif nodes are blue.

To conclude our study of ISP network motifs, in Figure 7 we show an example for how the motifs cover a specific AS topology (namely AS-4755).

6 Related Work

Our work is motivated by the virtualization trend in today's Internet and especially network virtualization. For a general introduction and a good survey, we refer the reader to [8]. Our model differentiates between a customer (e.g., a service provider requesting VNets) and a substrate provider (e.g., a physical infrastructure provider or a virtual network provider). In the terminology of [27], our customer is the SP, and the provider may either be the PIP or the VNP.

VNet Embedding. The embeddings of VNets is an intensively studied problem and there exists a large body of literature (e.g., [13, 14, 18, 31]), and there also exist distributed computing approaches [16] and online algorithms [5, 11]. Our work is orthogonal to this line of literature in the sense that we assume that an (arbitrary and not necessarily resource-optimal) embedding algorithm is given. Rather, we focus on the question of how the feedback obtained through these algorithms can be exploited, and we study the implications on the information which can be obtained about a provider's infrastructure.

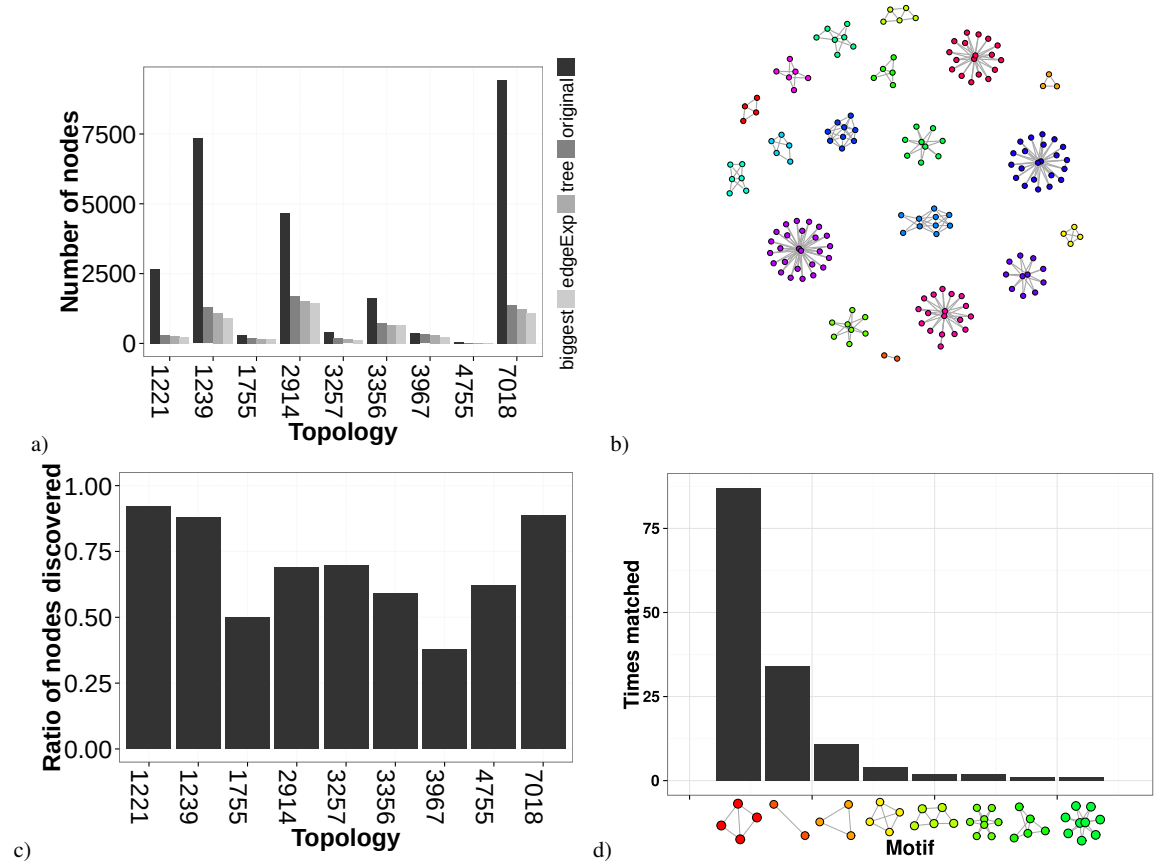

Fig. 6 Results of DICT when run on different Internet and power grid topologies. **a)** Number of nodes in different autonomous systems (AS). We computed the set of motifs of these graphs as described in Definition 6 and counted the number of remaining nodes after removing the nodes that : (i) belong to a tree structure at the "fringe" of the network, (ii) have degree 2 and belong to two-connected motifs, and finally (iii) are part of the largest motif. **b)** The 19-motif dictionary built from the dataset. **c)** The fraction of nodes that can be discovered with the dictionary. **d)** Most frequent motifs encountered in the dataset.

Topology Inference. Our work studies a new kind of topology inference problems. Traditionally, much graph discovery research has been conducted in the context of today's complex networks such as the Internet which have fascinated scientists for many years, and there exists a wealth of results on the topic. One of the most influential measurement studies on the Internet topology was conducted by the Faloutsos brothers [12], and their work has subsequently been intensively discussed both in practical [17] and theoretical [2] papers. The classic instrument to discover Internet topologies is *traceroute* [7], but the tool has several problems which renders the problem challenging. One complication of traceroute stems from the fact that routers may appear as stars (i.e., anonymous nodes), which renders the accurate characterization of Internet topologies difficult [1,23,30]. *Network tomography* is another important field of topology discovery. In network tomography, topologies are explored using pairwise end-to-end measurements, without the cooperation of nodes along these paths. This approach is quite flexible and applicable in various contexts, e.g., in social networks. For a good discussion of this approach as well as results for a routing model along shortest and second shortest paths see [4]. For example, [4] shows that for sparse random graphs, a relatively small number of cooperating participants is sufficient to discover a network fairly well.

Both the traceroute and the network tomography problems differ from our virtual network topology discovery problem in that the exploration there is inherently path-based while we can ask for entire virtual graphs.

Virtualization and Security. The benefits and threats of virtualization are extensively studied but still not well-understood. A complete review of the research is beyond the scope of this paper, and we refer

the reader to the recent literature, e.g., on virtual machine collocation attacks [25].

Graph Grammars and Mining. We decided to describe our algorithms using a graph grammar formalism, and indeed, some graph grammar problems share commonalities with our work. Graph grammars are a powerful tool to generate and characterize topologies, and we refer the interested reader to, e.g., [6]. More remotely, our work also has connections with *graph data mining* [29]. For instance, in [9], an algorithm is presented to search subsequences which can best compress an input graph based on a minimum description length principle. Although these algorithms pursue a different goal, the computationally-constrained beam search is reminiscent of some of our techniques as nodes are incrementally (and greedily) expanded.

Bibliographic Note. Our article builds upon our model introduced in a Brief Announcement at DISC [20]; the tree and cactus related results appeared at INFOCOM 2013 [21] and the dictionary was presented at NETYS 2013 [22].

7 Conclusion

This paper has initiated, from an algorithmic perspective, the discussion of topology discovery in network virtualization environments, and presented tight bounds for three important graph classes. We understand our results as a first step to shed light on possible security threats of the virtualization technology.

We find that the topology of typical sparse backbone networks such as tree or cactus graphs can be discovered relatively fast (request complexity $O(n)$). However, the motif-based topology discovery framework we sketched suggests that the request complexity for denser graphs is higher, as the number of graph knittings can grow combinatorially, and one has to resort to testing edges individually (after computing a spanning tree) which yields a quadratic request complexity.

Our work opens several interesting directions for future research. First, more work is needed to understand the implications of the framework on other important graph classes, such as different types of *planar graphs*: in order to beat the trivial $O(n^2)$ upper bound on the request complexity, additional dictionary properties must be exploited.

Moreover, in the context of TREE we have seen that if the host graph is not a tree, running TREE will simply result in a spanning tree. This raises the question whether similar spanning structures can be computed with incomplete motif sets, resulting in the "densest spanning structures" *given these motif sets*. For example, when applying CACTUS to a general graph, which fraction of edges will be discovered?

Finally, while our work so far has focused on discovering the *entire* topology, more specific graph properties may be inferred much more efficiently.

Acknowledgments

We would like to thank Georgios Smaragdakis from Telekom Innovation Laboratories for interesting discussions. Part of this work was performed within the Virtu project, funded by NTT DOCOMO Euro-Labs, and the Collaborative Networking project, funded by Deutsche Telekom AG. We would like to thank all our colleagues in these projects.

References

1. H. Acharya and M. Gouda. On the hardness of topology inference. In *Proc. ICDCN*, pages 251–262, 2011.
2. D. Achlioptas, A. Clauset, D. Kempe, and C. Moore. On the bias of traceroute sampling: or, power-law degree distributions in regular graphs. In *Proc. 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 694–703, 2005.
3. C. Ambühl, M. Mastrolilli, and O. Svensson. Inapproximability results for maximum edge biclique, minimum linear arrangement, and sparsest cut. *SIAM J. Comput.*, 40(2):567–596, Apr. 2011.
4. A. Anandkumar, A. Hassidim, and J. Kelner. Topology discovery of sparse random graphs with few participants. In *Proc. SIGMETRICS*, 2011.
5. N. Bansal, K.-W. Lee, V. Nagarajan, and M. Zafer. Minimum congestion mapping in a cloud. In *Proc. 30th PODC*, pages 267–276, 2011.
6. D. Caucal. Deterministic graph grammars. *Logic and Automata*, pages 169–250, 2008.
7. B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In *Proc. USENIX Annual Technical Conference (ATEC)*, 2000.
8. M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Elsevier Computer Networks*, 54(5), 2010.
9. D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Int. Res.*, 1:231–255, 1994.
10. J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34(3):313–356, Sept. 2002.
11. G. Even, M. Medina, G. Schaffrath, and S. Schmid. Competitive and deterministic embeddings of virtual networks. In *Proc. ICDCN*, 2012.
12. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proc. SIGCOMM*, pages 251–262, 1999.
13. J. Fan and M. H. Ammar. Dynamic topology configuration in service overlay networks: A study of reconfiguration policies. In *Proc. IEEE INFOCOM*, 2006.
14. C. Fuerst, S. Schmid, and A. Feldmann. Virtual network embedding with collocation: Benefits and limitations of preclustering. In *Proc. 2nd IEEE International Conference on Cloud Networking (CLOUDNET)*, November 2013.

15. A. Haider, R. Potter, and A. Nakao. Challenges in resource allocation in network virtualization. In *Proc. ITC Specialist Seminar on Network Virtualization*, 2009.
16. I. Houidi, W. Louati, and D. Zeghlache. A distributed virtual network mapping algorithm. In *Proc. IEEE ICC*, 2008.
17. A. Lakhina, J. Byers, M. Crovella, and P. Xie. Sampling biases in ip topology measurements. In *Proc. IEEE INFOCOM*, 2003.
18. J. Lischka and H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proc. ACM SIGCOMM VISA*, 2009.
19. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, 2008.
20. Y. A. Pignolet, S. Schmid, and G. Tredan. Brief announcement: Do vnet embeddings reveal isp topology? In *Proceedings of 26th International Symposium on Distributed Computing (DISC)*, 2012.
21. Y.-A. Pignolet, S. Schmid, and G. Tredan. Adversarial VNet Embeddings: A Threat for ISPs? In *IEEE INFOCOM*, 2013.
22. Y. A. Pignolet, S. Schmid, and G. Tredan. Request complexity of vnet topology extraction: Dictionary-based attacks. In *International Conference on Networked Systems (NETYS)*, May 2013.
23. Y. A. Pignolet, G. Tredan, and S. Schmid. Misleading Stars: What Cannot Be Measured in the Internet? In *Proc. DISC*, 2011.
24. J. M. Plotkin and J. W. Rosenthal. How to obtain an asymptotic expansion of a sequence from an analytic identity satisfied by its generating function. *J. Austral. Math. Soc. Ser. A*, 1994.
25. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. 16th ACM CCS*, pages 199–212, 2009.
26. G. Schaffrath, S. Schmid, and A. Feldmann. Optimizing long-lived cloudnets with migrations. In *Proc. IEEE/ACM UCC*, 2012.
27. G. Schaffrath, C. Werle, P. Papadimitriou, A. Feldmann, R. Bless, A. Greenhalgh, A. Wundsam, M. Kind, O. Maennel, and L. Mathy. Network virtualization architecture: Proposal and initial prototype. In *Proc. ACM SIGCOMM VISA*, 2009.
28. N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, 2004.
29. T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5:59–68, 2003.
30. B. Yao, R. Viswanathan, F. Chang, and D. Waddington. Topology inference in the presence of anonymous routers. In *Proc. IEEE INFOCOM*, pages 353–363, 2003.
31. S. Zhang, Z. Qian, J. Wu, and S. Lu. An opportunistic resource sharing and topology-aware mapping framework for virtual networks. In *Proc. IEEE INFOCOM*, 2012.

A Proof of Lemma 1

We prove the lemma for undirected graphs. The extension to directed graphs is straight-forward.

A poset structure $(S, \preceq)$ over a set S requires that $\preceq$ is a (*reflexive*, *transitive*, and *antisymmetric*) order which may or may not be partial. To show that $(\mathcal{G}, \mapsto)$, the embedding order defined over a given set of graphs $\mathcal{G}$, is a poset, we examine the three properties in turn.

Reflexivity (for each $G \in \mathcal{G}$, $G \mapsto G$): By using the identity mapping $\pi : G = (V, E) \to G = (V, E)$ which embeds each node and link to itself, the claim is proved.

Transitivity (for all $A, B, C \in \mathcal{G}$, if $A \mapsto B$ and $B \mapsto C$ then $A \mapsto C$): Let π_1 denote the embedding function for $A \mapsto B$ and let π_2 denote the embedding function for $B \mapsto C$, which must exist by our assumptions. We will show that then also a valid embedding function π exists to map A to C. Regarding the node mapping, we define π_V as $\pi_V := \pi_{1V} \circ \pi_{2V}$, i.e., the result of first mapping the nodes according to π_{1V} and subsequently according to π_{2V}. We first show that π_V is a valid mapping from A to C as well. First, for all $v_A \in V_A$, $\pi(v_A)$ maps v_A to a single node in V_C, fulfilling the first condition of the embedding (see Definition 1). Ignoring relay capacities (which is studied together with the conditions on the links below), Condition (ii) of Definition 1 is also fulfilled since the mapping π_{1V} ensures that no node in V_B exceeds its capacity, and can hence safely be mapped to V_C. Let us now turn our attention to the links. We use the following mapping π_E for the edges. Note that π_{1E} maps a single link e to an entire (but possibly empty) path in B and π_{2E} then maps the corresponding links e' in B to a walk in C. We can transform any of these walks into paths by removing cycles; this can only lower the resource costs. Since π_{1E} maps to a subset of E_B only and since π_{2E} can embed all edges of B, all link capacities are respected up to relay costs. However, note also that by the mapping π_1 and for relay costs $\epsilon > 0$, each node $v_B \in V_B$ can either not be used at all, be fully used as a single endpoint of a link $e_A \in E_A$, or serve as a relay for one or more links. Since both end-nodes and relay nodes are mapped to separate nodes in C, capacities are respected as well. Conditions (iii) and (iv) hold as well.

Antisymmetry (for all $A, B \in \mathcal{G}$, $A \mapsto B$ and $B \mapsto A$ implies $A = B$, i.e., A and B are isomorphic): First observe that if the two networks differ in size, i.e., $|V_A| \neq |V_B|$ or $|E_A| \neq |E_B|$, then they cannot be embedded to each other: Without loss of generality, assume $|V_A| > |V_B|$, then since nodes of V_A of cannot be split into multiple nodes of V_B (cf Definition 1), there exists a node $v_A \in V_A$ to which no node from V_B is mapped. This however implies that node $\pi_1(v_A) \in V_B$ must have available capacities to host also v_A, contradicting our assumption that nodes cannot be split in the embedding. Similarly, if $|E_A| \neq |E_B|$, we can obtain a contradiction with the single path argument. Thus, not only the total number of nodes and links in A and B must be equivalent but also the total amount of node and link resources. So consider a valid embedding π_1 for $A \mapsto B$ and a valid embedding π_2 for $B \mapsto A$, and assume $|V_A| = |V_B|$ and $|E_A| = |E_B|$. It holds that π_1 and π_2 define an isomorphism between A and B: Clearly, since $|V_A| = |V_B|$, π_1 and π_2 define a permutation on the vertices. Without loss of generality, consider any link $\{v_A, v'_A\} \in E_A$. Then, also $\{\pi_1(v_A), \pi_1(v'_A)\} \in E_B$: $|\{\pi_1(v_A), \pi_1(v'_1)\}| = 0$ would violate the node capacity constraints in B, and $|\{\pi_1(v_A), \pi_1(v'_A)\}| > 1$ requires $|E_B| > |E_A|$.

B Proof of Lemma 2

(i) From Definition 1, $A \mapsto B$ defines a mapping π from V_A to V_B and from E_A to E_B, which implies that the subgraph where A is embedded on B constitutes a minor of B. Since $\epsilon > 0.5$ every node of V_B in $\pi(A)$ is used exactly once. Contracting the edges on paths of $\pi(e)$ for each $e \in E_A$ thus results in A which proves the claim. (ii) For smaller ϵ there are graphs which can be embedded into other graphs H although they are not minors of H, e.g., the forbidden minor K_5 (i.e., the clique with five nodes) can be embedded in some planar graphs, see Figure 8 for an ex-

ample. (iii) The opposite direction of (i) does not hold, as can be seen in the following example demonstrating that not all minors of a substrate are embeddable. Given ϵ, let $r = \lceil 1/\epsilon \rceil + 1$ and consider a r-ary tree of height 2 with r^2 leaves. A star graph with r^2 leaves is a minor of this tree, however, it cannot be embedded in the tree because the relay nodes cannot support r paths without exceeding their capacities and hence there are not enough paths to all leaves.

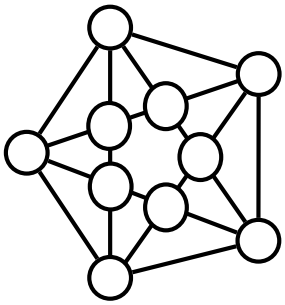

Fig. 8 Planar graphs are K_5 minor free, but the planar substrate H depicted in this figure a K_5 can be embedded if $\epsilon \leq 0.5$.

C Proof of Lemma 4

Let $v \in G$. Clearly, if there exists $u \in M$ such that $v = \pi(u)$, then v's capacity is fully used. Otherwise, v was added by Rule 2. Let a, b be the two nodes of G between which Rule 2 was applied, and hence $\{\pi^{-1}(a), \pi^{-1}(b)\} \in E_M$ must be a motif edge. Observe that for these nodes' degrees it holds that $\deg(a) = \deg(\pi^{-1}(a))$ and $\deg(b) = \deg(\pi^{-1}(b))$ since Rule 2 never modifies the degree of the old nodes in the host graph G. Since links are of unit capacity, each substrate link can only be used once: at a at most $\deg(a)$ edge-disjoint paths originate, which yields a contradiction to the degree bound; the relaying node v has a load of ϵ.

D Proof of Lemma 5

By contradiction assume $i \in \max_{v \in V_D}(v \mapsto S)$ and i is not a subsequence of S (written $i \not\prec S$). Since D *covers* S we have $S \in V_D^\star$ by definition.

Since D is a dictionary and since $i \mapsto S$, we know that $S \notin Q_i$. Thus, $S \in D^\star \backslash Q_i$: S has a subsequence of at least one word in R_i. Thus there exists $k \in R_i$ such that $k \prec S$. If $k = i$ this implies $i \prec S$ which contradicts our assumption. Otherwise it means that $\exists j \in \Gamma^+(i)$ such that $j \mapsto k \prec S$, which contradicts the definition of $i \in \max_{v \in V_D}(v \mapsto S)$ and thus it must hold that $i \prec S$.

E Proof of Lemma 6

We present a procedure to construct such a dictionary D. Let $\mathcal{M}_n$ be the set of all motifs with n nodes of the graph family $\mathcal{H}$. For each motif $m \in \mathcal{M}_n$ with x possible attachment point pairs (up to isomorphisms), we add x dictionary words to V_D, one for each attachment point pair. The resulting set is denoted by V_M. For each sequence of $V_M^\star$ with at most n nodes, we add another word to V_D (with the un-used attachment points of the first and the last subword). There is an edge $e \in E_D$ if the transitive reduction of the embedding relation with context includes an edge between two words. We now prove that D is a dictionary, i.e., it is robust to composition. Let $i \in V_D$. Observe that R_i contains all compositions of words with at most n nodes in which i can be embedded. Consequently, no matter which sequences are in $\overline{R}_i^\star$, it holds that v_i cannot be embedded in sequences in Q_i, and the robustness condition is satisfied. Since H has n vertices, and since D contains all possible motifs of at most n vertices, D covers H.

F Proof of Theorem 3

We first prove that the claim is true if H forms a motif sequence (without edge expansion). Subsequently, we study the case where the motif sequence is expanded by Rule 2, and finally tackle the general composition case.

Discovery of motif sequences: Due to Lemma 5, it holds that for w chosen when Line 1 of *findMotifSequence*() is executed for the first time, S is partitioned into three subsequences S_1, w and S_2. Subsequently *findMotifSequence*() is executed on each of the subsequences $S' \in \{S_1, S_2\}$ recursively if $C \mapsto S'$, i.e., if the subsequences are not empty. Thus *findMotifSequence*() computes a decomposition as described in Corollary 3 recursively. As each of the words used in the decomposition is a subsequence of S, and as *findMotifSequence*() does not stop until no more words can be added to any subsequence, it holds that all nodes of S will be discovered eventually. In other words, $\pi^{-1}(u)$ is defined for all $u \in S$.

As a next step we assume $S' \neq S$ to be the sequence of words obtained by DICT to derive a contradiction. Since $S' := H'$ is the output of algorithm DICT and is hence embeddable in H: $S' \mapsto S$, there exists a valid embedding mapping π. Given $u, v \in V(S)$, we denote by $E^{\pi^{-1}}(S')$ the set of pairs $\{u, v\}$ for which $\{\pi^{-1}(u), \pi^{-1}(v)\} \in E(S')$. Now assume that S and S' do not lead to the same resource reservations. Hence there are some inconsistencies between the substrate and the output of algorithm DICT: $\Phi = \{\{u, v\} \in E(S) \backslash E^{\pi^{-1}}(S') \cup E^{\pi^{-1}}(S') \backslash E(S)\}$. With each of these "conflict" edges, one can associate the corresponding word $W_{u,v}$ (resp. $W'_{u,v}$) in S (resp. S'). If a given conflict edge spans multiple words, we only consider the words with the highest index as defined by DICT. We also define $i_{u,v} = r(W_{u,v})$ (resp. $i'_{u,v} = r(W'_{u,v})$). Since S' and S are by definition not isomorphic, $i'_{u,v} \neq i_{u,v}$.

Let $j = \max_{(u,v) \in \Phi}(i_{u,v})$ be the index of the greatest word embeddable on the substrate containing an inconsistency, and j' be the index of the corresponding word detected by DICT.

(i) Assume $j > j'$: a lower order motif was erroneously detected. Let J^+ (and J^-) be the set of dictionary entries that are detected before (after) $D[j]$ (if any) in S by DICT. Observe that the words in J^+ were perfectly detected by DICT, otherwise we are in Case (ii). We can decompose S as an alternating sequence of words of J^+ and other words using Corollary 3 : $S = T_1 J_1(a_1) T_2 \dots T_k$ with $J_i(a_i) \in (J^+)^\star$ and attachment points a_i and $T_i \in (J^-)^\star$. As the words in J^+ are the same in S', we can write $S' = T'_1 J_1 T'_2 \dots T'_k$ (using Corollary 3 as well).

Let T be the sequence among $T_1, \dots, T_k$ that contains our misdetected word $D[j]$, and T' the corresponding sequence in S'. Observe that $T' \mapsto T$ since the words J_i cut the sequences of S and S' into subsequences T_i, T'_i that are embeddable. Observe that $D[j] \mapsto T$ since T contains it. Note that in the execution of *findMotifSequence*() when $D[j']$ was detected the higher indexed words had been detected correctly by DICT in

previous executions of this subroutine. Hence, $T_<$ and $T_>$ cannot contain any words leading to edges in Φ. We deduce that $j' \neq \arg\max_x(D[x] \mapsto T)$ since $j = \arg\max_x(D[x] \mapsto T)$ and $j' < j$ which contradicts Line 1 of *findMotifSequence*().

(ii) Now assume $j' > j$: a higher order motif was erroneously detected. Using the same decomposition as step (i), we define J'^+ as the set of words perfectly detected, and therefore decompose S and S' as sequences $S = T_1 J'_1 T_2 \dots J'_{k-1} T_k$ and $S' = T'_1 J'_1 T'_2 \dots J'_{k-1} T'_k$ with $J'_i \in (J'^+)^\star$ and the property that each $T'_i \mapsto T_i$.

Let T' be the sequence among $T'_1, \dots, T'_k$ that contains our misdetected word $D[j']$, and T the corresponding sequence in S. Since $D[j'] \prec T'$, $D[j'] \mapsto T'$. Recall that $T \in V_D^\star \backslash (J'^+)^\star$. We again consider two subcases: $D[j'] \prec T$: T contains some occurrences of the word $D[j']$, but DICT detected a wrong number of such occurrences. Using Corollary 3, we again decompose T as $T = R_1 D[j'] R_2 \dots R_k$ with $R_i \in (J'^-)^\star$. Let R be the sequence $R_1, \dots, R_k$ containing $D[j]$. We have $D[j'] \mapsto R$ and $R \in (J'^-)^\star$, which contradicts the robustness property of D. Now, consider that T has no occurrences of the word $D[j']$: $T \in V_D^\star \backslash (J'^+ \cup \{D[j']\})^\star$, that is $T \in (J'^-)^\star$ and $D[j'] \mapsto T$, which again contradicts the robustness property of D.

The same arguments can be applied recursively to show that conflicts in ϕ of smaller indices cannot exist either.

Expanded motif sequences. As a next step, we consider graphs that have been extended by applying node insertions (Rule 2) to motif sequences, so-called *expanded* motif sequences: we prove that if H is an expanded motif sequence S, then algorithm DICT correctly discovers S. Given an expanded motif sequence S, replacing all degree-2 nodes with an edge connecting their neighbors unless a cycle of length three would be destroyed, leads to a unique pure motif sequence T, $T \mapsto S$. For the corresponding embedding mapping π it holds that $V(S) \setminus \pi(T)$ is exactly the set $\mathcal{R}$ of removed nodes. Applying *findMotifSequence*() to an expanded motif sequence discovers this pure motif sequence T by using the nodes in $\mathcal{R}$ as relay nodes. All nodes in $\mathcal{R}$ are then discovered in *edgeExpansion*() where the reverse operation node insertion is carried out as often as possible. It follows that each node in S is either discovered in *findMotifSequence*() if it occurs in a motif or in *edgeExpansion*() otherwise.

Combining expanded sequences. Finally, it remains to combine the expanded sequences. Clearly, since motifs describe all parts of the graph which are at least 2-connected, the graph remaining after collapsing motifs cannot contain any cycles: it is a tree. However, on this graph DICT behaves like TREE, but instead of attaching chains, entire sequences are attached to different nodes. Along the unique sequence paths between two nodes, DICT fixes the largest words first, and the claim follows by the same arguments as used in the proofs for tree and cactus graphs.

G Proof of Lemma 7

The request complexity of Line 1 of In the depth-first traversal, there is exactly one path between the chain C and a word $v = D[i]$ in V_D. DICT issues a request for at most all the outgoing neighbors of the nodes this path. After v has been found, the highest j where $H' v\ (T_<)\ \text{BF}\ (v^j)\ \text{AF}\ (T_>) \mapsto H$ has to be determined. To this end, another $j+1$ requests are necessary. Thus the maximum of $cost(v) + j$ over all word $v \in V_D$ determines the request complexity.

H Proof of Theorem 4

Each time Line 1 of *findMotifSequence*() is called, either at least one new node is found or no other node can be embedded between the current sequences (one request is necessary for the latter result). If one or more new nodes are discovered, the request complexity can be amortized by the number of nodes found: If v is the maximal word found in Line 1 of *findMotifSequence*(), then v is responsible for at most $cost(v)$ requests due to Lemma 7. If it occurs more than once at this position, only one additional request is necessary to discover even more nodes (plus one superfluous request if no more occurrences of v can be embedded there). Amortizing the request number over the number of discovered nodes results in Δ requests. All other requests are due to *edgeExpansion*(e) where additional nodes are placed along edges. Clearly, these costs can be amortized by the number of edges in H: for each edge $e \in E(H)$, at most two embedding requests are performed (including a "superfluous" request which is needed for termination when no additional nodes can be added).