Compressing IP Forwarding Tables: Realizing Information-theoretical Space Bounds and Fast Lookups Simultaneously

Attila Kőrösi*, János Tapolcai†, Bence Mihálka†, Gábor Mészáros†, Gábor Rétvári†
*MTA-BME Information Systems Research Group, Budapest University of Technology and Economics, korosi@tmit.bme.hu
†MTA-BME Future Internet Research Group, Budapest University of Technology and Economics
{tapolcai, mihalka, retvari}@tmit.bme.hu, gabor.meszaros@ericsson.com

***Abstract*—The Internet routing ecosystem is facing compelling scalability challenges, manifested primarily in the rapid growth of IP packet forwarding tables. The forwarding table, implemented at the data plane fast path of Internet routers to drive the packet forwarding process, currently contains about half a million entries and counting. Meanwhile, it needs to support millions of complex queries and updates per second. In this paper, we make the curious observation that the entropy of IP forwarding tables is very small and, what is more, seems to increase at a lower pace than the size of the network. This suggests that a sophisticated compression scheme may effectively and persistently reduce the memory footprint of IP forwarding tables, shielding operators from scalability matters at least temporarily. Our main contribution is such a compression scheme which, for the first time, admits both the required information-theoretical size bounds and attains fast lookups, thanks to aggressive level compression. Although we find the underlying optimization problem NP-complete, we can still give a lightweight heuristic algorithm with firm approximation guarantees. This allows us to squeeze real IP forwarding tables, comprising almost 500,000 prefixes, to just about 140–200 KBytes of memory within a factor of 2–3 of the entropy bound, so that forwarding decisions take only 8–10 memory accesses on average and updates are supported efficiently. Our compression scheme may be of more general interest, as it is applicable to essentially any prefix tree.**

***Index Terms*—IP forwarding, prefix trees, data compression**

I. Introduction

The number of routers, end-hosts, and Autonomous Systems that together comprise the Internet is increasing at a fast pace, and there is no sign of this trend slowing down any time soon. Correspondingly, it is widely recognized that the Internet routing ecosystem is facing significant long-term scaling challenges [1]. Beyond the natural expansion of the network, there are various additional factors that contribute to the rapid growth of the routed IP address space, like the wide-scale use of non-aggregatable provider-independent addresses and site multi-homing, prefix deaggregation for inbound traffic engineering, fragmentation of the address space due to the depletion of the allocatable IPv4 address pool and the spreading of IPv6, etc [2]–[4]. Naturally, this takes its toll on the Internet routing infrastructure, manifested as ever higher volumes of BGP signaling load at the control plane and rapidly expanding IP forwarding tables at the data plane. While the control plane challenge might not pose as significant of a concern [5], [6] as it was originally believed [1], data plane scalability has remained mostly an unsolved issue this far.

The IP forwarding table (Forwarding Information Base, FIB) lies at the heart of packet forwarding in IP routers. The FIB is essentially a giant database, associating to each individual routed IP address prefix the identifier of the next-hop router along the best path towards that prefix. The FIB is consulted on a packet-by-packet basis, which amounts to tens of millions of complex longest prefix match queries per second, and its content needs to be modified several thousands times per second during BGP update storms.

Easily, the larger the routed address space the more entries the FIB maintains, and this expansion has been anything but slow lately. According to our measurements as of May, 2014, IPv4 FIBs have gained $25,000$ new records just in the last 6 months, boosting FIB size to a whopping half a million entries. Correspondingly the FIB, as implemented in the Linux kernel [7] for instance, has grown from 24 Mbytes to more than 32 Mbytes in size. This expansion rate greatly outpaces the growth of fast cache available on commodity hardware, let alone commercial routers' expensive and hardly upgradeable embedded SRAMs, and this is beginning to present a critical performance bottleneck in the data plane performance [8]. Just downloading a FIB of this size to the Linux kernel takes almost 5 minutes, with similar control plane to data plane delays reported in commercial IP network gear [9].

Scalability concerns notwithstanding, exploding FIB sizes cause further headache to operators. Expanding fast memory on router line cards increases silicon footprint, heat production, and power budget, and forces operators into frequent and costly upgrades [10], [11]. Factor in the increasing popularity of network virtualization, which means that today a single physical router hardware needs to host many IP virtual routing instances, each with its own largish FIB, side-by-side [12]. Hence, even if *Moore's Law* will save the day in the long run (which may [13] or may not [1] be the case), growing FIB size still jeopardizes the very profitability of service providers by skyrocketing the CAPEX/OPEX associated with maintaining and operating the network infrastructure.

Strikingly, it has been reported recently that the information-

/978-1-4799-6204-4/14$31.00 ©2014 IEEE

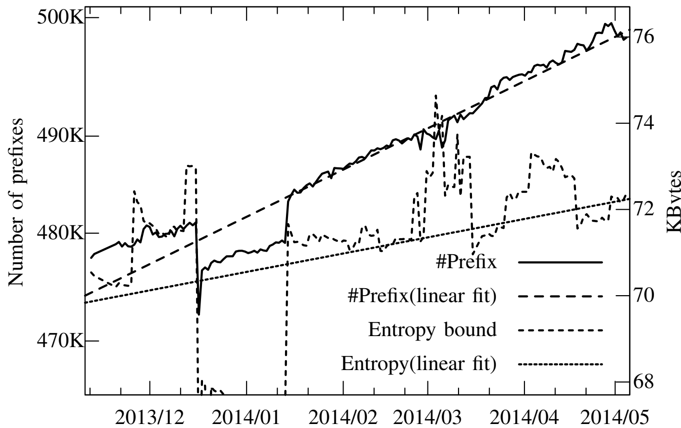


Fig. 1: Number of IPv4 prefixes and information-theoretical FIB entropy bound as observed on `rtr.bme.hbone.hu` during 6 months in 2013 and 2014.

theoretical entropy of IP forwarding tables is surprisingly low, implying that *IPv4 FIBs can be compressed remarkably efficiently* [14], [15]. In particular, it was shown that an ordinary DFZ (Default Free Zone) FIB, when compressed using a sophisticated encoder, needs only 1–1.5 bits(!) of storage space per IP address prefix, and still supports millions of longest prefix matches and hundreds of thousands of updates per second. What is more, as the compressed FIB occupies smaller space it can move closer to the CPU in the cache hierarchy in a software router, or may fit entirely into the line card SRAM of a hardware router, and accordingly *lookups on the compressed form are often even faster than on the uncompressed one*.

In this paper, we make the further observation that *over time the entropy of IP FIBs seems to grow slower than the number of prefixes*. We have downloaded[1] two dozen IP FIBs every day during 6 months from operational IP routers located in the US academic IP backbone `Internet2` and its Hungarian counterpart, the `HBONE`. Fig. 1 depicts the number of IP prefixes as seen at one of the core `HBONE` routers, along with the information-theoretical entropy bound. The entropy bound seems to grow at a slow pace, by a mere 2 KBytes during this half year interval, and this trend appears to be fairly general across the FIB instances we have examined, with DFZ FIBs compressing slightly better than those containing a default gateway. We conclude that *an entropy-bounded FIB compressor may mask the expansion of the address space over time, effectively shielding operators from Internet routing scalability concerns*.

FIB compression is an extensively researched field, with many interesting proposals on the table [16]–[29]. To the best of our knowledge though, none of these come with stringent information-theoretic space bounds, which would be needed to provide predictable storage size and performance guarantees persistently. The only exception seems to be [14], where a compressed re-invention of the conventional prefix tree data structure is introduced and shown to satisfy certain entropy-constrained space bounds. The basic idea is to eliminate recurrent sub-structures from the prefix tree yielding a compressed *prefix DAG* (Directed Acyclic Graph), similarly to how redundant sub-strings are eliminated by the Lempel-Ziv text compression scheme [30]. Unfortunately, the resultant prefix DAG is fundamentally *binary*, which transforms into 32 memory accesses during an IPv4 lookup in the worst-case (and way worse with IPv6), hardly scaling to gigabit line-speeds.

Our main contribution in this paper is the *application of the level-compression technique to prefix DAGs*, in order to eliminate excess interior nodes from the FIB representation thereby cutting down the number of memory accesses per lookup. This leads us to a *level-compressed prefix DAG* scheme that *attains both information-theoretically minimal space and improved lookup performance at the same time*[2]. A rudimentary level-compressed serialization scheme for prefix DAGs was already proposed in [14]. Herein, we show how to do level-compression *optimally*. Our level-compressed prefix DAGs occupy 25% smaller storage space than binary ones in general, while sustaining 5–10% higher lookup speed. Meanwhile, obtaining the compressed form remains similarly simple and updates can be supported with the same computational complexity. Level-compressed prefix DAGs, furthermore, can be created at each router locally, without the need to notify neighbors, plus they remain fully compatible with contemporary router ASICs [32], simplifying deployment to a software upgrade. Our compression technique applies to essentially any area where prefix trees are used, and therefore may be of generic interest beyond the scope of IP routing.

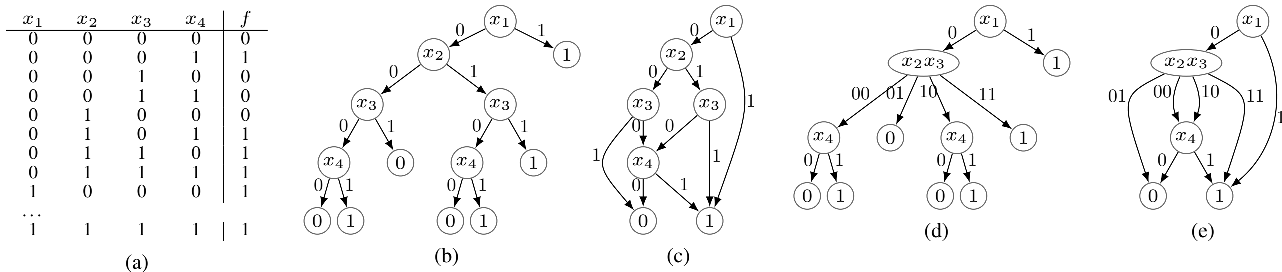
The paper is structured as follows. After some background on prefix trees (Section II) and stating the main problem formally (Section III), first we analyze the computational complexity of level-compressing prefix DAGs (Section IV). Curiously, we find that, contrary to most prefix tree compression problems, the combined problem is NP-complete. Then, we present an optimal Integer Linear Program and a very fast heuristic algorithm and we show that the heuristics simultaneously delivers an approximate solution *and* a tight optimality gap characterizing its quality, making it unnecessary to solve the ILP in the first place (Section V). Finally, we evaluate the performance of our algorithm in extensive numerical studies, we measure the lookup and update performance on a prototype Linux kernel implementation (Section VI), and then we draw the conclusions (Section VII).

II. Prefix trees

Prefix trees, or *tries*, are amongst the most ubiquitous data structures in computer science. Given a set of input strings and some data associated with each string, a trie allows to access

[1] We have created a website we dubbed the *Internet Routing Entropy Monitor* to publish browsable daily statistics and downloadable data sets for the community, see http://lendulet.tmit.bme.hu/fib_comp.

[2] Earlier research, although termed under similar name [17], [31], was aimed at a different purpose.



x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
...				
1	1	1	1	1

Fig. 2: Representations of the Boolean function $f(x_1, x_2, x_3, x_4) = x_1 + x_2x_3 + \overline{x}_3x_4$: (a) truth table; (b) reduced binary decision tree; (c) binary decision diagram; (d) level-compressed decision tree (d); and (e) level-compressed prefix DAG.

and update the data for any string in time proportional to the length of that string [33]. Applications span a wide range, from associative memories keyed over any ordered sequence, to lookup tables, sparse bitmaps, XML DOMs, word completion libraries and dictionaries, full-text indexes, etc. Prefix trees can even be used for efficient sorting. Unfortunately, the memory footprint can become huge, ruining performance on modern CPUs due to cache misses. Correspondingly, space-efficient trie representation has become a focal problem lately [34].

A popular application of prefix trees is *binary decision trees* [35]. Suppose we are given the Boolean function $f(x_1, x_2, x_3, x_4) = x_1 + x_2x_3 + \overline{x}_3x_4$ and we would like to evaluate f very fast. A way to achieve this is to organize the truth table of f (see Fig. 2a) into a tree and a set of associated labels, with each leaf node corresponding to a key whose label gives the result of evaluating f on that key. A (reduced) *binary decision tree* is essentially such a trie where identically labeled leaves descending from a common parent are contracted (Fig. 2b). Then, evaluating an input equals following the path in the trie traced out by the input values, first x_1, then x_2, x_3, and finally x_4. The lookup operation terminates when the search arrives into a labeled leaf. Note that edge labels are not stored in the trie but instead implicit in the ordering of the pointers laid out in memory. Thus, a node's position uniquely determines the associated prefix.

There are various ways to cut down the memory footprint of binary decision trees (of which reduction is immediately one option). For one, merging isomorphic subgraphs, taking into account the labels on the leaves, yields a more space-efficient *binary decision diagram* (Fig. 2c), also called a *binary prefix DAG*. Whereas our sample trie representation contains 13 nodes and 12 pointers, the prefix DAG holds only 7 nodes and 10 pointers. Meanwhile, lookup complexity (and the actual lookup algorithm) do not change.

Trie compaction can even improve lookup performance in certain cases. Instead of eliminating redundant portions of the trie, one could go on and eliminate excess interior nodes to obtain a *level-compressed binary decision tree* (see Fig. 2d). Here, nodes can represent any succession of variables and pointers are stored for each input combination that can show up. In our example, for the input $x_1x_2x_3x_4 = 0110$ the function f is evaluated as follows: first, at the root we check x_1 and since its value is 0 we descend to the left sub-trie, then we evaluate x_2 and x_3 simultaneously yielding the index 11, following which we arrive to a leaf node so we immediately return the corresponding label (i.e., 1). Not just that the number of nodes and pointers drop by 2, but lookup now goes in only 3 steps instead of 4, provided that we can evaluate multiple consecutive variables in constant time, which is usually true on modern CPU architectures.

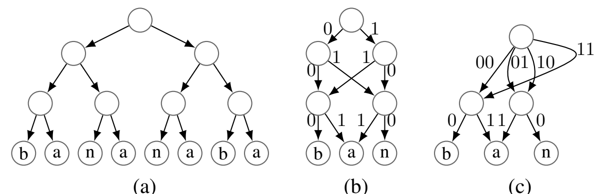
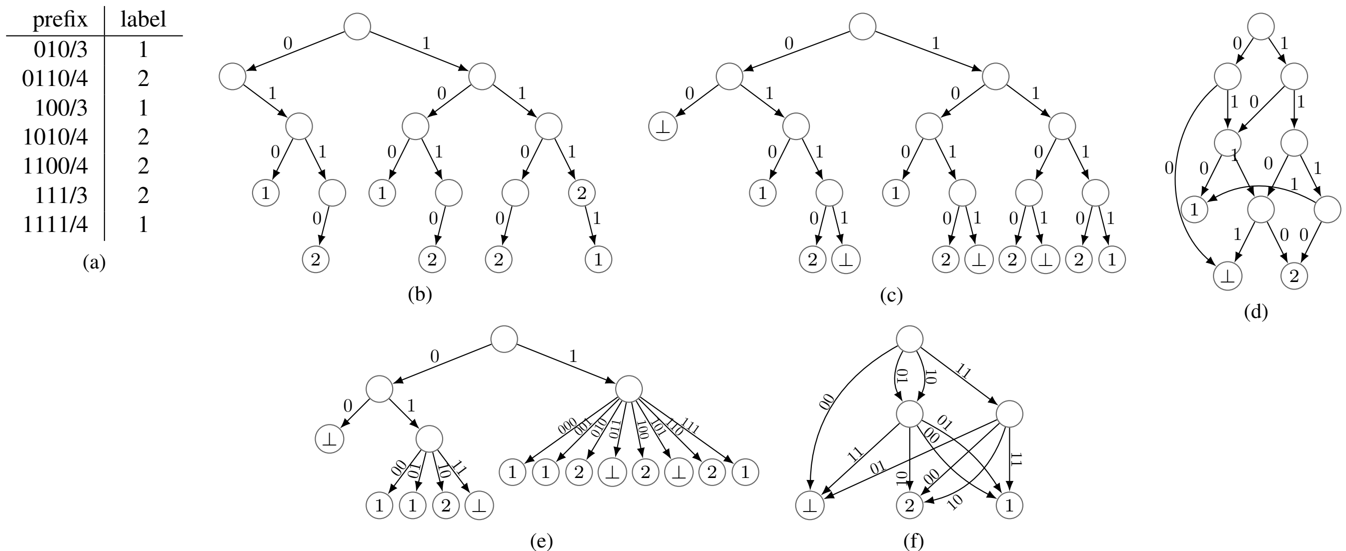


Fig. 3: Trie representations for the string "bananaba": (a) binary prefix tree; (b) binary prefix DAG; and (c) level-compressed prefix DAG. The third character of the string can be accessed by looking up the key $3 - 1 = 010_2$.

Another appealing application of prefix trees is *compressing and indexing textual data* [36]. The idea is to map the string to be compressed to a complete binary prefix tree, with each leaf representing the symbol at the position corresponding to its key (see Fig. 3a) and then convert this into a binary prefix DAG (see Fig. 3b). Again, the space reduction is obvious. This is so much so that the space to store a binary prefix DAG has been recently shown to be proportional to the Shannon-entropy of the input string [14], essentially realizing the theoretical minimum of a zero-order string compressor [30] up to a small constant factor and lower order terms. This still lags behind optimal compression algorithms, like Huffman coding or LZW [30], but crucially *a prefix DAG also admits fast in-place random access and update* to the string, which traditional compressors do not support without explicit decompression. Level-compression could also be applied to save on access time and storage space, but in this example this would give a trivial one-level trie basically corresponding to an array.

The most important application of prefix trees within the context of this paper is implementing lookup tables for IP packet forwarding. As mentioned earlier, an IP FIB consists of hundreds of thousands of entries, each entry specifying an



prefix	label
010/3	1
0110/4	2
100/3	1
1010/4	2
1100/4	2
111/3	2
1111/4	1

Fig. 4: Representations of an IP forwarding table: (a) tabular representation, specifying for each prefix and prefix length pair the corresponding next-hop label; (b) unnormalized prefix tree; (c) normalized (leaf-pushed) binary trie with the invalid label $\perp$; (d) binary prefix DAG; (e) level-compressed trie; and (f) level-compressed prefix DAG.

address-prefix-to-next-hop-label association telling to which neighbor to pass on a packet (see Fig. 4a). Multiple entries may match a particular address, like in our example both the $111/3 \to 2$ and $1111/4 \to 1$ entries match an address starting with 1111. In such cases the longest prefix match rule applies and the entry fitting on the greatest number of bits (from the MSB) takes preference. In our case, this means that the second entry overrides the first one, resulting label 1 for the lookup.

Storing the FIB as a binary prefix tree (see Fig. 4b) allows to find the longest matching prefix efficiently, in time proportional to the width of the address space. Unfortunately, this form is not really space-efficient. A simple way of space reduction is *leaf-pushing* [19], yielding a *normalized* prefix-free format after removing all less-specifics (Fig. 4c). Binary prefix DAGs (Fig. 4d) also lend themselves as a space-efficient FIB representation, inheriting the information-theoretical storage size guarantees from string compression [14]. Level-compressed prefix trees (Fig. 4e), on the other hand, have been advocated for their vastly improved lookup performance [7], [16], [37], [38]. Unfortunately, *level-compressed prefix trees lack the theoretically justified space-efficiency of binary prefix DAGs, which in turn lack improved access and update times of level-compressed tries*. So far, satisfying both requirements simultaneously has been an open research challenge.

III. Problem formulation

In this paper, we propose to combine level-compression and prefix DAGs into a new compressed trie representation. The resultant *level-compressed prefix DAG* (lcDAG) for decision diagrams is given in Fig. 2e, for strings in Fig. 3c, and for IP FIBs in Fig. 4f, respectively. Observe that in each case both the number of nodes/pointers *and* the number of steps needed to perform a lookup drop. Later, we shall show by extensive numerical evaluations that these advantages manifest on real instances as well. Thusly, our main concern now is to compress a binary prefix tree into a level-compressed prefix DAG.

Definition 1. *Given a set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ defined on an alphabet Ω and for each $S \in \mathcal{S}$ a label l_S taking its value from an alphabet Σ, a prefix DAG (D, l) is tuple of a rooted DAG $D(V_D, E_D)$ and a label function $l : V_D \mapsto \Sigma$. Each node $v \in V_D$ has $|\Omega|^{k_v} : k_v \in \mathbb{N}$ children and each edge $e = (v, u) \in E_D$ represents a unique sequence of k_v symbols $c_e \in \Omega^{k_v}$, such that:*

- *the root r of D represents the empty prefix; and*
- *$S \in \mathcal{S}$, if and only if there is a node $v \in V_D$ and a directed $r \to v$ path $p = \{e_1, e_2, \ldots, e_k\} : e_i \in E_D$, so that $S = c_{e_1} c_{e_2} \ldots c_{e_k}$ and $l(v) = l_S$.*

A prefix tree, or a trie, is a prefix DAG (D, l) where D is a rooted tree. In this paper, we shall concentrate on the *bitwise* case where the input are 0–1 strings: $\Omega = \{0, 1\}$. Note that any input can be represented this way by binary encoding the symbols in Ω. We shall also assume that the input is *prefix-free*, that is, no string in $\mathcal{S}$ is a prefix of some other string in $\mathcal{S}$. In addition, we presume that the input is *proper*, meaning that if $Xc \in \mathcal{S}$ for some prefix X and $c \in \Omega$, then the prefix Xq also appears in $\mathcal{S}$ for *all* $q \in \Omega$ (i.e., there is some $S \in \mathcal{S}$ so that Xq is a prefix of S). We have seen the leaf-pushing algorithm as a simple example of how to pre-process the input so that all these properties are fulfilled (but see also [36]). For strings S that are introduced by the pre-processing algorithm into $\mathcal{S}$ we set the *invalid label* $l_S = \perp$, for a special symbol $\perp \in \Sigma$. Finally, we make the technical assumption that the label alphabet Σ is not too large, say, $|\Sigma|$ is $O(\text{polylog}\, n)$ with $n = |\mathcal{S}|$, or even $O(1)$. The assumption is trivially true for binary decision diagrams, and it holds for many practically relevant

string compression [36] and FIB compression instances [39].

The prefix DAG for such a bitwise, prefix-free, proper input has the property that every node v is either a labeled leaf or it is an interior node with 2^{k_v} outgoing edges, where 2^{k_v} is called the *stride size* of v for some $k_v > 0$ integer. Such prefix DAGs are called *leaf-labeled* and *proper*, and also *binary* if $k_v = 1$ uniformly at all nodes or *multibit* otherwise.

If we seek our data structure in the form of a *binary prefix tree* (T, l), then we can build (T, l) in $O(n \log n)$ time and $O(n \log n)$ space, of which accessing or updating any element is possible in $O(\log n)$ steps. Our task is then to improve these bounds. That is, we want to compress (T, l) to an alternative form (D, f) so that

(i) (D, f) is *semantically equivalent* with (T, l), denoted by $(D, f) \sim (T, l)$, meaning that for any binary string $S \in \{0, 1\}^W$, where W is the length of the longest string in $\mathcal{S}$, the lookup operation yields precisely the same label on (D, f) and (T, l); and

(ii) the amount of memory $M(D, f)$ needed to store (D, f) is minimal and meets firm information-theoretical storage size bounds [14], [15], [34], described below[3].

Proposition 1. *Given a proper, binary, leaf-labeled trie* (T, l) *on* n *leaves labeled on an alphabet* Σ *of size* $\sigma = |\Sigma|$*:*

- *the* information-theoretic lower bound *for storing* (T, l) *is* $2n + n \lg \sigma$ *bits*[4]*; and*
- *the* zero order entropy *of* (T, l) *is* $2n + nH_0$ *bits, where*

$$H_0 = \sum_{c \in \Sigma} p_c \log_2 \left({}^{1}/{}_{p_c} \right)$$

denotes the Shannon-entropy of the leaf-label probabilities $p_c = {}^{n_c}/{}_{n}$, $n_c = |\{v \in V_T : l(v) = c\}|$.

A. Compressing into a Binary Prefix DAG

One way to reduce the memory footprint of a prefix tree is to convert it into a *binary prefix DAG*. Correspondingly, the MINBINPREFIXDAG problem asks for compressing (T, l) into a binary prefix DAG (D, f). The following claims apply [14], [15].

1) MINBINPREFIXDAG can be solved in $O(n)$ time;
2) (D, f) can be stored on at most $4n \lg \sigma + o(n)$ bits, under the assumption that the length of each string in $\mathcal{S}$ is W and $n = 2^W$ (i.e., (T, l) is a *complete* binary trie);
3) if $H_0 < \lg \sigma$, then the expected size of (D, f) is $(6 + 2 \lg \frac{1}{H_0} + 2 \lg \lg \delta) H_0 n + o(n)$ bits;
4) accessing any element in (D, f) can be done in $O(\log n)$ steps; and
5) updating (D, f) can be done in $O((1 + {}^{1}/{}_{H_0}) \log n)$ time.

In summary, a binary prefix DAG compresses a prefix tree to at most roughly four times the information-theoretic limit (so it is a *compact data structure*) and six times the entropy (so it is also a *compressed data structure)* [34]), without hurting lookup and update performance in any ways.

[3]Note that the original paper [14] states the space bounds erroneously. In this paper we give the correct bounds; see also [15].

[4]The notation $\lg x$ is shorthand for $\lceil \log_2(x) \rceil$.

B. Compressing into a Level-compressed Prefix Tree

Alternatively, we could seek the output in the form of a *level-compressed trie*. The corresponding form of the trie compression problem is called MINLEVCOMPTRIE. The following observations apply [37], [40]:

1) MINLEVCOMPTRIE can be solved in $O(n \log n)$ time;
2) lookup on (D, f) terminates in $O(\log n)$ steps as worst-case and in $O(\log \log n)$ steps on a broad class of inputs [40]; and
3) update terminates in $O(\log n)$ steps provided that the maximum stride size is $O(1)$.

In summary, solving MINBINPREFIXDAG yields a data structure with appealing entropy-bounded storage size guarantees, while MINLEVCOMPTRIE results blazingly fast $O(\log \log n)$ lookup performance [7].

C. Compressing into a Level-compressed Prefix DAG

Our main observation in this paper is that by combining level-compression and binary prefix DAGs into a new trie compression scheme, one can realize the advantages of both techniques simultaneously.

Definition 2. MINLEVCOMPDAG*: given a proper, leaf-labeled, binary trie* (T, l) *find a proper, leaf-labeled, level-compressed prefix DAG* (D, f)*, so that* $(D, f) \sim (T, l)$ *and* $M(D, f)$ *is minimal.*

With a slight abuse of notation, we shall denote by MINLEVCOMPDAG(k) the decision version of the problem where the task is to decide whether a prefix DAG (D, f) exists with $M(D, f) \leq k$ for some positive integer k.

In what follows, we generally assume that the storage size $M(D, f)$ is dominated by the number of (constant size) pointers, thus omitting the $\sigma \lg \sigma$ bits needed to describe the labels. This assumption is mostly in line with the literature [16], [40]. For simplicity, we initially focus on the static version of the problem, and we shall return to the question of updates only later in Section VI.

IV. COMPLEXITY

Next we turn to our main contributions. First, we analyze the computational complexity of MINLEVCOMPDAG.

So far, we have seen that level-compression and *binary* prefix DAG compression alone are both tractable in (roughly) linear time. Interestingly, their combination appears difficult.

Theorem 1. MINLEVCOMPDAG(k) *is NP-complete.*

The proof can be found in the Appendix. The transformation is from the maximal independent set problem in 3-regular (cubic) graphs [41].

This finding is not completely unexpected. The example in Fig. 4 demonstrates that a naive "trie-threading" approach [14], [42] that would simply identify and share isomorphic sub-tries directly on the level-compressed tree fails to find the correct result. The other way around, that is, level-compressing the binary prefix DAG would not work either. Instead, one has to, at each node, deliberately balance between whether to merge

isomorphic children or compress a level, taking into account that over-expanding the stride of a node may destroy otherwise shareable trie instances below, while shrinking the stride too much introduces excess intermediate levels. It turns out that making this decision optimally is difficult.

V. A HEURISTIC ALGORITHM

It seems that solving MINLEVCOMPDAG to optimality is hard. Instead of shooting for optimality, therefore, below we present fast heuristic algorithms. These not only output an approximate solution rapidly but also supply an upper bound as well as a lower bound on the memory consumption of the optimal solution, this way providing a thorough characterization of the quality of the solution retrieved.

First, we need some notation. We are given a proper, leaf-labeled, binary trie (T, l) as input. Denote the nodes of T by V_T, let r be the root node, and let L_T be the set of leaves and I_T the set of interior nodes. Let $n = |L_T|$. For any $u \in V_T$ let $h(u)$ denote the height of the sub-tree rooted at u, let $d(u)$ be the level of u, $d(r) = 0$, and let $\mathcal{P}_i(u)$ denote the i-th parent of u for any $i \in \{1, \ldots, d(u)\}$ (i.e., $\mathcal{P}_1(u)$ is the immediate parent, $\mathcal{P}_2(u)$ is the parent's parent, etc.). We further assume that we have a relation τ available that partitions the nodes of T into equivalence classes based on whether the sub-tries descending from those nodes are isomorphic. Namely, we treat nodes $u, v \in V_T$ isomorphic, denoted as $\tau(u) = \tau(v)$, if either *(i)* $u, v \in L_T$ and $l(u) = l(v)$ or *(ii)* $u, v \in I_T$ and each of the left and right children are pairwise isomorphic. In fact, τ could be thought of as a function that maps from V_T to the node set V_D of a *binary* prefix DAG $D(V_D, E_D)$. As a matter of fact, $D(V_D, E_D)$ is the smallest binary prefix DAG representation for (T, l) and so finding τ is equivalent to solving MINBINPREFIXDAG, which can be done in linear time. We denote as τ^{-1} the inverse of τ, which to a DAG node v orders the set of tree nodes isomorphic to v.

We take off from the trie level-compression algorithm of Sahni et al. [40]. Consider the below dynamic program to solve MINLEVCOMPTRIE.

$$x_u = \min_{i \in \{1..h(u)\}} \left(2^i + \sum_{w \in V_T : \mathcal{P}_i(w) = u} x_w \right) \qquad \forall u \in I_T \quad (1)$$

$$x_u = 0 \qquad \forall u \in L_T \quad (2)$$

Here, x_u marks the amount of memory consumed by the sub-trie rooted at node u in the optimal level-compressed trie. Easily, the leaves store no pointers, and if an interior node u chooses the stride 2^i then the size of its sub-trie equals the number of pointers descending from it (i.e., 2^i) plus the memory consumed by all the i-level children. Then, we seek the stride that minimizes this expression, which can be done for the whole tree in a bottom-up traversal. Unfortunately, this algorithm only works for tries but not for prefix DAGs.

Our heuristic algorithm is a slight modification of this scheme. The idea is that if multiple isomorphic tree nodes choose the same stride size then these can be shared in the level-compressed prefix DAG and thereby contribute jointly to the memory consumption of that DAG node. Correspondingly, we assign a non-negative *weight* parameter λ_u^i to each $u \in V_T$ and each potential stride $i \in \{1..h(u)\}$ and we impose the *correctness* condition requiring that the weight of isomorphic tree nodes at stride i adds up to 2^i:

$$\sum_{u \in \tau^{-1}(v)} \lambda_u^i = 2^i, \quad \lambda_u^i \geq 0 \qquad \forall v \in V_D, \forall i \in \{1..h(v)\} \ . \quad (3)$$

Choose some λ_u^i so that the correctness condition holds and collect these into a single vector λ. Then, for each such λ we define the following dynamic program DP(λ):

$$x_u = \min_{i \in \{1..h(u)\}} \left(\lambda_u^i + \sum_{w \in V_T : \mathcal{P}_i(w) = u} x_w \right) \qquad \forall u \in I_T \quad (4)$$

$$x_u = 0 \qquad \forall u \in L_T \quad (5)$$

The following observations are now immediate.

(i) If we apply no sharing of nodes (i.e., we set τ as the identity function) and the correctness condition (3) holds, then the dynamic programs DP(λ) and (1)–(2) coincide.

(ii) DP(λ) can be solved in a single bottom-up traversal in $O(n \log n)$ steps similarly to [40].

(iii) From this we can recover a valid prefix DAG as follows: We cycle through each tree node $u \in V_T$ and we note the depth i at which (4) takes its minimum. If any isomorphic tree node w has previously chosen the same stride, we relabel the parent of u to w. Otherwise we insert u into the DAG at stride i. This can be done in $O(n \log n)$ steps.

In summary, our dynamic program DP(λ) supplies a valid, although not necessarily optimal, prefix DAG for any "correct" λ in $O(n \log n)$ steps. The actual choice of λ is arbitrary and we get different algorithms for each weight setting. The heuristic nature of the algorithm lies in that we do not know the weight setting that solves MINLEVCOMPDAG, not even whether such weights exist at all. Curiously, we can still give a characterization of the quality of the solution obtained and, as it turns out, this holds over *any* correct setting of λ.

Let (D^*, f^*) be the optimal prefix DAG that solves MIN-LEVCOMPDAG, let (D_λ, f_λ) denote the prefix DAG obtained by solving DP(λ), and let $x_r(\lambda)$ denote the value of the x_r variable that belongs to the root r in this solution.

Theorem 2. *For any λ that satisfies* (3)*:*

$$x_r(\lambda) \leq M(D^*, f^*) \leq M(D_\lambda, f_\lambda) \ .$$

Accordingly, if we solve DP(λ) we not only get the prefix DAG (D_λ, f_λ) but also a lower bound $x_r(\lambda)$ and an upper bound $M(D_\lambda, f_\lambda)$ on the optimal objective, providing a qualitative measure on the goodness of the solution. This measure is usually taken in the form of an *optimality gap*

$$\mu(\lambda) = \frac{M(D_\lambda, f_\lambda) - x_r}{x_r} \ . \quad (6)$$

If $\mu(\lambda)$ is small, say, under 1–2%, we know that what we have obtained is a prefix DAG very close to the optimal one, while larger values indicate a poor quality result. This is why such

lower and upper bounds are extremely useful in constructing approximation algorithms of verifiable performance [43].

The rest of this section is devoted to proving Theorem 2. The part $M(D^*, f^*) \leq M(D_\lambda, f_\lambda)$ is obvious as both (D^*, f^*) and (D_λ, f_λ) are feasible in MINLEVCOMPDAG but the former is also optimal. What remained to be seen is the part $x_r(\lambda) \leq M(D^*, f^*)$.

The proof is built on defining an Integer Linear Program (ILP), of which (D^*, f^*) will be a feasible solution and whose linear programming relaxation will deliver an upper bound on $x_r(\lambda)$. The ILP contains the following variables: let $s_u^i, u \in V_T, i \in \{1..h(u)\}$ (respectively, $z_v^i, v \in V_D, i \in \{1..h(v)\}$) be an integer variable for each tree node u (for each DAG node v), so that $s_u^i > 0$ ($z_v^i > 0$) if and only if node u (resp. v) is optimized to stride size 2^i. The ILP itself is as follows.

$$z^* = \min \sum_{v \in V_D} \sum_{i=1}^{h(v)} 2^i z_v^i \tag{7}$$

$$\sum_{i=1}^{d(u)} s_{\mathcal{P}_i(u)}^i \leq \sum_{i=1}^{h(u)} s_u^i \qquad \forall u \in V_T \setminus \{r\} \tag{8}$$

$$1 \leq \sum_{i=1}^{h(r)} s_r^i \tag{9}$$

$$s_u^i \leq z_{\tau(u)}^i \qquad \forall u \in V_T, \forall i \in \{1..h(u)\} \tag{10}$$

$$s_u^i \geq 0, s_u^i \in \mathbb{Z}, z_{\tau(u)}^i \in \mathbb{Z} \qquad \forall u \in V_T, \forall i \in \{1..h(u)\} \tag{11}$$

Here, constraint (8) requires that a node must show up in the solution if any of its i-level parents chooses just the stride size 2^i. Condition (9) bootstraps the chain by requiring that the root actually appear in the result. So far, we have only dealt with tree nodes[5]. The constraints that link the tree solution to the DAG are (10), guaranteeing that a DAG node appears in the solution if any of the isomorphic tree nodes is set at the same stride. The rest of the constraints set the domains for the variables. Finally, the objective function sums the number of pointers in the DAG, counting each isomorphic tree node that happens to be optimized to the same stride size, and thereby being merged into a single DAG node, only once. Denote the optimal objective function value by z^*.

It is not immediately obvious but the above ILP, if solved, delivers precisely the optimal level-compressed prefix DAG. To actually see this, we would need to tediously work out that at optimality all variables are binary and $z_v^i = \min_{u \in \tau^{-1}(v)} s_u^i$. Instead, herein we confine ourselves to the following simple claim (which we will not need later anyways).

Observation 1. *The ILP* (7)–(11) *solves* MINLEVCOMPDAG.

To actually prove the theorem, we need a weaker result.

Lemma 1. (D^*, f^*) *is feasible in the ILP* (7)–(11).

Proof. This is rather straightforward to show: for each node $v \in V_{D^*}$, if v is set to stride size 2^i then set $z_v^i = 1$ and set $\forall u \in \tau^{-1}(v) : s_u^i = 1$, and set all variables to zero otherwise. This way, (10) and (11) trivially hold, and (8) and (9) also fulfill because $s_u^i = 1 : u \in V_T$ makes up a tree. □

[5] In fact, the ILP this far with the objective $\min \sum_{u \in V_T} \sum_{i=1}^{h(u)} 2^i s_u^i$ would give a (rather inefficient) algorithm for MINLEVCOMPTRIE [16], [40].

Corollary 1. $z^* \leq M(D^*, f^*)$.

Now, consider the linear programming relaxation of (7)–(11) obtained by substituting constraints (11) as follows:

$$s_u^i \geq 0, s_u^i \in \mathbb{R}, z_{\tau(u)}^i \in \mathbb{R} \qquad \forall u \in V_T, \forall i \in \{1..h(u)\} \ .$$

Letting x_u to denote the dual variables associated with constraints (8)–(9) and (with again abusing the notation a bit) λ_u^i to denote the dual variables for (10), the dual problem for the relaxed linear program is easily seen to be:

$$\hat{z} = \max \ x_r \tag{12}$$

$$x_u \leq \ \lambda_u^i + \sum_{w \in V_T : \mathcal{P}_i(w) = u} x_w \qquad \forall u \in I_T, \forall i \in \{1..h(u)\} \tag{13}$$

$$\sum_{u \in \tau^{-1}(v)} \lambda_u^i = 2^i \qquad \forall v \in V_D, \forall i \in \{1..h(v)\} \tag{14}$$

$$x_u \geq 0, \lambda_u^i \geq 0 \qquad \forall u \in V_T, \forall i \in \{1..h(u)\} \tag{15}$$

Lemma 2. *For any* λ *that satisfies* (3) *and any* $x_u(\lambda)$ *that solves* DP(λ), λ *and* $x_u(\lambda)$ *are feasible in* (13)–(15).

Proof. We observe that (14) is just identical to the correctness condition (3) so it is satisfied by assumption, (13) trivially holds by (4), and again (15) holds by assumption. □

Denoting the optimal objective function value of (13)–(15) by $\hat{z}$, we conclude:

Corollary 2. $x_r(\lambda) \leq \hat{z}$.

Putting this all together we get the following proof.

Proof of Theorem 2. We see that $x_r(\lambda) \leq \hat{z}$ for any "correct" setting of λ by Corollary 2; $\hat{z} \leq z^*$ because by strong duality $\hat{z}$ equals the optimal objective value of the primal, which is in turn smaller than z^* by being a relaxation thereof; $z^* \leq M(D^*, f^*)$ by Corollary 1; $M(D^*, f^*) \leq M(D_\lambda, f_\lambda)$ was seen earlier; thus we get $x_r(\lambda) \leq M(D^*, f^*) \leq M(D_\lambda, f_\lambda)$ which completes the proof. □

As a final note, we emphasize again that this holds for *any* choice of the weights λ as long as non-negativity and the correctness condition (3) are respected. For the rest of this paper, we use the simple weight setting that distributes the cumulative weight 2^i equally between the contributing nodes:

$$\lambda_u^i = \frac{2^i}{|\tau^{-1}(\tau(u))|} \qquad \forall u \in V_T, \forall i \in \{1..h(u)\} \ . \tag{16}$$

VI. PERFORMANCE EVALUATION

We subjected our heuristic approximation scheme and the resultant level-compressed prefix DAGs to comprehensive performance evaluations. Our aims were to verify whether and to what extent the theoretical storage size guarantees hold, to study the quality of the approximation in terms of the optimality gap worked out above, to examine how often

TABLE I: Results for different FIB-compression algorithms on real IP FIBs. The columns indicate the name of the FIB instance, number of prefixes N, and number of next-hops σ in it; Shannon-entropy of the next-hop label distribution H_0; information-theoretic limit I and entropy bound E for storing the trie ([KByte]); and size (M, [KByte]) and compression efficiency ν, for the `fib_trie`, level-compressed binary trie (lcTrie), binary prefix DAG (bDAG), and level-compressed prefix DAG (lcDAG) FIB implementations.

							`fib_trie`		lcTrie		bDAG		lcDAG	
		N	σ	H_0	I	E	M	ν	M	ν	M	ν	M	ν
HBONE	`bme`	499,211	89	1.20	196	71	32198	451.05	635.95	8.91	203.72	2.85	162.05	2.27
	`szeged`	499,236	87	1.20	196	71	32198	450.75	636.06	8.90	203.76	2.85	162.09	2.27
	`vh1`	499,143	207	2.34	407	185	32197	173.60	1157.48	6.24	503.22	2.71	393.34	2.12
	`vh2`	499,302	101	1.20	196	72	32202	450.31	636.49	8.90	204.13	2.85	162.36	2.27
Internet2	`atlanta`	14,312	93	1.91	38	14	1017	71.82	110.75	7.82	48.65	3.44	41.11	2.90
	`houston`	14,305	101	1.12	38	14	1016	71.93	109.47	7.75	49.00	3.47	41.49	2.94
	`kansas`	14,335	101	1.06	38	14	1019	73.31	110.43	7.95	47.38	3.41	39.99	2.88
access	`taz`	410,513	4	0.97	94	56	26698	474.62	519.23	9.23	172.95	3.07	138.75	2.47
	`access(d)`	444,513	28	1.61	206	95	28713	302.07	869.80	9.15	252.88	2.66	201.23	2.12
	`access(v)`	2,986	3	0.99	3	2	192	88.99	15.23	7.06	8.26	3.83	7.16	3.32
	`mobile`	21,783	10	1.62	1	0	290	655.67	2.47	5.59	1.27	2.88	1.19	2.69

level-compression should be re-applied, and to justify that the complexity of the lookup operation indeed improves.

The inputs were as follows. For forwarding table compression we used real IPv4 FIBs obtained from operational Internet routers. First, we used 4 DFZ instances from the `HBONE` (`bme`, `szeged`, `vh1`, `vh2`) and 3 smaller FIB instances from the `Internet2` (`atlanta`, `houston`, `kansas`). These FIBs were downloaded on 29 April, 2014, and represent the state-of-the-art in IP routing tables as of spring 2014. We also included 4 FIBs taken from [14], [15] in our evaluations. These instances come from various service provider access networks: `taz` and `access(d)` are FIBs from the DFZ, while the smaller `access(v)` and `mobile` FIBs contain default routes. We also experimented with compressing and indexing textual data. Here, we considered a zero-order source that spits out Bernoulli-distributed symbols taken from a simple binary alphabet. The first symbol appears with probability p and the second with probability $1-p$, and varying p allows to adjust the entropy between 0 and 1. String length was set to 2^{17}. Finally, we examined compressing decision trees of Boolean functions arising from simple random 3SAT instances. Due to space constraints, however, we cannot give the results here, but we note that the conclusions were essentially the same.

We implemented each trie compression method discussed in the paper. We used the algorithm (1)–(2) of Sahni *et al.* to solve MinLevCompTrie (lcTrie); MinBinPrefixDAG was solved with the algorithm from [14] (bDAG); and we used the algorithm outlined in Section V for MinLevCompDAG (lcDAG) with λ taken as of (16) and optimality gap μ as (6). For reference we included in the evaluations the standard Linux kernel FIB implementation `fib_trie`, an adaptive level- and path-compressed multibit trie-based FIB code [7].

After compression, we serialized the resultant FIBs into a binary blob, which was then fed into a custom kernel module embedded into the Linux IP forwarding engine. The storage scheme is as simple as it can get: after a header and some metadata comes the next-hop table, which is then followed sequentially by the prefix tree/prefix DAG nodes, first the nodes with the largest stride and then subsequently the nodes with smaller and smaller strides all the way to the leaves. Children of a node are laid out sequentially in the file. The id space is divided into two parts: the first part is used to identify each node uniquely, while the last couple of ids are allocated for next-hop labels. The number of bits needed to encode the ids is taken as the base-2 logarithm of the size of the id space obtained this way, and our home-grown packed-array implementation [44] was used to encode these ids into the serialized form. This scheme is universal in that it supports any proper leaf-labeled input, i.e., it can be used to encode each of the lcTrie, bDAG, and lcDAG forms. The FIB sizes reported later correspond to the size of this serialized blob. Compression efficiency was measured as the fraction of the size of the compressed form to the theoretical entropy bound. Easily, the smaller the compression efficiency the better.

The experiments were carried out on an Intel Core i5 processor, 3092.696 MHz, with 128 kByte L1 and 1024 kByte L2 cache, and 6 MB L3 Cache, with 32 bit word size. We aimed to measure the raw performance, so we used only a single CPU core. Note that this platform is different from the one used in [14] and the serialization schemes differ too, hence the difference in the results.

A. Storage Size

The results for IP FIB compression are given in Table I. Our first observation is that, even if the binary form already admits very compact memory representation, the true power of prefix DAGs manifests itself only with level-compression. *Our algorithm was able to compress DFZ IPv4 FIBs to only about* 140–200 *KBytes of memory, sporting a compression efficiency of roughly* 2. The only exception is the `vh1` FIB instance, which contains extraordinarily many next-hops due to operational reasons, this way doubling the entropy and hence storage space. Interestingly, this instance produced the best lcDAG compression efficiency amongst the DFZ FIBs.

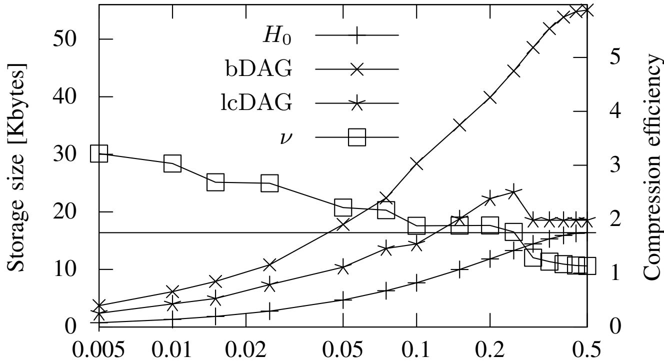


Fig. 5: Size and compression efficiency over strings with Bernoulli distributed symbols as the function of parameter p. The solid line marks the information-theoretic limit.

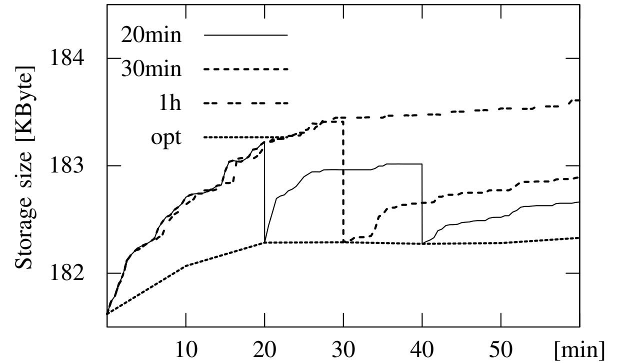


Fig. 6: lcDAG size for `access(d)` over a one hour BGP update sequence when re-executing level-compression after every 20, 30, or 60 minutes, or running it continuously (opt.).

Contrast this with the latest reported FIB sizes for >$400K$ prefixes from the literature, ranging from 780 KBytes (DXR, [27]) to 1.2 Mbytes (SMALTA, [20]). This is in line with our results: binary prefix DAGs are in the 200–300 Kbyte range for DFZ FIBs, exhibiting a compression efficiency of more than 3, while compression efficiency is around 10 for level-compressed prefix trees and rises to the hundreds with `fib_trie`. As it is usual in data compression, larger FIBs compress better than smaller ones.

We also observed the optimality gap μ of our heuristic algorithm as defined by (6). As it turns out, our simple heuristics gives an incredibly good approximation for IP FIBs, since the *optimality gap was only* 1–2% in each case. This indicates that the level-compressed prefix DAGs we obtained are very close to optimum, making it completely unnecessary to shoot for an optimal solution of the otherwise intractable MINLEVCOMPDAG problem. Instead, our dynamic program DP (4)–(5) runs in only about 70–110 milliseconds in general, depending on the size of the input, the entropy, and other parameters. This indicates that even if we were to implement updates to the DAG by re-optimizing it from scratch after every modification, we could still support tens or even hundreds of updates per second (but see below).

We repeated the evaluations on textual input instead of IP FIBs. Fig. 5 gives the results for the Bernoulli-source, when varying parameter p from 0 to 0.5. Here, compression efficiency is almost perfect for a uniform source, and it gradually degrades for more biased symbol distributions (this behavior is confirmed theoretically and empirically in [14]) but for all $p \leq 0.1$ we can still squeeze the string below the information-theoretic limit. Indeed, it is somewhat startling to realize that we can get efficient compression just by storing a string as a plain *directed graph*. Optimality gap was similarly low as in the case of FIB compression, and running times were also in the millisecond range.

B. Updates

As it turns out, adding a new entry to a prefix DAG or deleting/modifying an existing one is far from trivial, not because it is a DAG *per se* but rather due to that it is normalized (leaf-pushed). Normalization is indispensable to reliably identify shareable sub-tries and obtain good compression, but also makes updates expensive as a label modification close to the root may cause relabeling essentially all leaves. The authors in [14] find a simple workaround: they observe that at higher levels shareable sub-tries are rare and so normalization is unnecessary, and by carefully controlling the level below which leaf-pushing is applied they simultaneously attain entropy-bounded space and close to optimal updates.

Even though this trick extends from binary prefix DAGs to level-compressed ones transparently, we do not use it in this paper. Rather, we used a simple update routine that blatantly renormalizes the entire sub-trie affected by a label modification. The reason is that we found even this naive update implementation to work reasonably for the most common use case: when updates arrive from BGP. As BGP originated FIB updates are heavily biased towards longer prefixes, the size of the sub-tries needed to be re-packed per update is usually very small, and so even our simple routine works fast.

We modeled a real BGP router's workload as follows. We downloaded a BGP log from RouteViews, and we treated all prefix announcements as generating a FIB update with a next-hop selected randomly according to the next-hop distribution of the FIB. The resultant sequence consists of $30,000$ individual updates, corresponding to roughly one hour of BGP churn, with a mean prefix length of 21.8. We ran our update routine on this input, in each run re-applying our heuristic level-compression routine after every 20 minutes, 30 minutes, or 1 hour worth of BGP updates. The results for the `access(d)` FIB are given in Fig. 6.

We expected the size of the prefix DAG to gradually deteriorate with time as the updates ruin the delicate level-compressed structure. Interestingly, the results did not confirm this expectation: the prefix DAG seems mostly insensitive to the interval with which level-compression is applied, with about 1.5 KBytes of extra space accumulating initially but mostly saturating afterwards. In addition, we could reach anything between $13,000$–$17,000$ updates per second on average

TABLE II: Lookup benchmark results for randomly selected IP addresses on real IP FIBs. Serialization is used with native 32 bit arrays. The columns indicate, for each of the `fib_trie`, level-compressed binary trie (lcTrie), binary prefix DAG (bDAG), and level-compressed prefix DAG (lcDAG) FIB implementations, the mean height ($\overline{h}$) and maximum height ($h_{\max}$) of the prefix tree/prefix DAG, the lookup performance in million lookups per second (`mpps`), the number of CPU cycles spent per lookup (`#CPU`), and mean cache-miss rate per 10000 lookup operations (`cm`).

		fib_trie					lcTrie					bDAG					lcDAG				
		$\overline{h}$	$h_{\max}$	mpps	#CPU	cm	$\overline{h}$	$h_{\max}$	mpps	#CPU	cm	$\overline{h}$	$h_{\max}$	mpps	#CPU	cm	$\overline{h}$	$h_{\max}$	mpps	#CPU	cm
HBONE	bme	2.43	8	4.07	758	23900	6.62	13	10.39	297	2145	25.85	31	10.69	289	1926	9.61	15	10.92	283	1903
	szeged	2.43	8	4.09	756	23583	6.62	13	10.40	297	2135	25.57	31	10.69	289	1960	9.49	15	10.92	283	1902
	vh1	2.43	8	4.09	754	23407	5.56	13	10.11	306	2533	25.52	31	9.92	311	2359	7.56	15	10.24	301	2247
	vh2	2.43	8	4.05	763	24463	6.62	13	10.40	297	2137	25.86	31	10.69	289	1965	9.55	15	10.88	284	1902
Internet2	atlanta	3.37	10	5.87	526	2442	6.31	14	9.94	311	1803	29.45	31	10.65	290	1842	11.04	17	11.15	277	1834
	houston	3.37	10	5.86	527	2476	6.15	14	9.99	309	1837	28.72	31	10.64	290	1839	11.12	16	11.11	278	1839
	kansas	3.37	10	5.87	526	2496	6.04	15	10.21	303	1796	29.44	31	10.65	290	1836	10.89	16	11.25	274	1823
access	taz	2.42	6	4.45	694	20735	6.53	13	10.57	292	2081	21.00	31	10.73	288	1882	7.75	16	10.96	282	1874
	access(d)	2.44	8	4.25	726	21941	6.06	14	9.25	334	2417	28.85	31	10.81	286	2105	10.96	16	11.00	281	1982
	access(v)	5.53	9	12.45	248	1798	10.31	15	12.94	239	1817	20.67	31	13.40	230	1828	10.67	18	14.83	208	1846
	mobile	6.68	9	12.33	250	1813	9.70	14	14.13	218	1776	26.44	31	17.67	174	1828	11.00	16	22.56	137	1832

for the DFZ FIB instances, and even higher for the smaller FIBs. This means that we finish with one hour of BGP workload in about 2 seconds(!). While this performance seems more than sufficient for the BGP use case, it may not be sufficient for others. For such cases, adopting the algorithm from [14] seems plausible, which would guarantee orders of magnitudes faster update operations on any update pattern.

C. *Lookup Performance*

Finally, we subjected our level-compressed prefix DAGs to extensive lookup tests on a real software router. Our implementation runs inside the Linux IP forwarding engine, hijacking the kernel's network stack to send IP lookup requests to our custom kernel module that in turn consults the serialized blob as generated by the FIB compressor routines. We used the standard Linux network micro-benchmark tool `kbench` on our custom module [45], which calls the FIB lookup function in a tight loop and measures the number of CPU cycles burnt and the execution time with nanosecond precision. We modified `kbench` to take a sequence of 1 million uniformly distributed IPv4 addresses in the interval $[0, 2^{32} - 1]$. The route cache was disabled. We also observed the average and maximum height of the compressed prefix trees and DAGs, as these parameters together set the number of memory accesses needed to trace down a key to a leaf. Finally, we measured the number of CPU cache misses during each lookup by monitoring the `cache-misses` CPU performance counter with the `perf(1)` tool. For the experiment, we set the serialization to native 32 bit arrays in order to eliminate any performance toll of unaligned memory accesses in packed arrays. We then repeated the experiment over packed arrays, and also over a packet trace in the "CAIDA Anonymized Internet Traces 2012" data set [46] instead of random addresses, but the outcome was essentially the same. Results for the random IP address lookup benchmark are given in Table II.

We observe that, thanks to aggressive path- and level-compression, `fib_trie` reduces the height of FIBs significantly, resulting only about 2–3 memory accesses per lookup for DFZ FIBs. We see, however, that essentially each of these memory accesses generates a cache miss event[6], confining the CPU to spend numerous empty cycles waiting for the result from the main memory to become available. This limits performance at about 4 million lookups per second. Level-compressed prefix trees take much smaller space, leading to an order of magnitude fewer cache misses and two- to three-fold increase in lookup performance. Binary prefix DAGs, on the other hand, imply even fewer cache misses, but are slower due to the vast interior nodes needed to be traversed during every lookup. Our level-compressed prefix DAGs though seem to unify the advantages of lcTrie and bDAG: the average height and the memory footprint are small enough to warrant the best lookup performance amongst the examined FIB compression schemes. The performance gain, as compared to lcTrie, is about 2–6% on DFZ instances and 10–60% on smaller FIBs. The reason for this moderate gain seems to be our evaluation platform that is just too powerful: with the exception of `fib_trie` essentially all the rest of the FIBs fit into the L2 cache entirely, smoothing the differences. We expect that in a hardware router with limited SRAM size or on very large FIBs obtained by aggregating multiple virtual routers' forwarding tables [12], [47] the difference would be way more substantial.

VII. Conclusions

With the advent of big data, processing massive volumes of information is becoming increasingly compelling. Compressed data structures allow to squeeze large amounts of data into size-constrained fast memory sidestepping the obligatory space-time trade-off: the smaller the data structure the closer it drifts to the CPU in the cache hierarchy, so we get even better performing applications. Compressed data structures, however, have yet to permeate the networking community as of now.

In this paper, we take on the quest to turn this trend around. We were driven by the observations that *(i)* real-

[6]Note that cache miss rate is specified per 10000 lookup operations.

life IP forwarding tables are characterized by surprisingly low information-theoretical entropy, suggesting that they may lend themselves readily to a compression algorithm; *(ii)* FIB entropy has grown at a low pace over the last couple of months, indicating that the compression algorithm has the potential to mask the effects of Internet growth from service providers, at least temporarily; and *(iii)* the lookup performance offered by the existing schemes that can take advantage of this and compress to entropy bounds is generally not adequate to support multi-gigabit line speeds [14].

To overcome this issue, we have combined two well-known trie compaction techniques, level-compression and binary prefix DAGs. Even though the underlying problem turned out intractable, our heuristic algorithm proved an efficient approximation scheme. We found that the resultant level-compressed prefix DAGs fit below roughly 2–3 times the entropy bound in size and still support lookup and update very fast.

Our current update routine is mostly optimized for BGP workloads. Our plan is to adopt the scheme from [14] to generalize to arbitrary use cases. Furthermore, the equal weight setting heuristic (16) is completely intuitive. It would be interesting to theoretically confirm this setting in the future, find better ones, or even incorporate weights into a Lagrangian framework to solve the problem optimally. On a longer term, we see intriguing opportunities to extend our work to compressing general forwarding tables [48] and packet classifiers [49], [50], virtual routers' shared forwarding tables [12], [47], or general labeled trees [34].

ACKNOWLEDGEMENTS

This research was partially supported by High Speed Networks Laboratory (HSN Lab) and the Hungarian Scientific Research Fund under grant No. OTKA 108947. J. T. was supported by the project TÁMOP - 4.2.2.B- 10/1–2010-0009 and G. R by the OTKA/PD-104939 grant. The authors wish to thank the NIIF Institute and the Internet2 consortium for providing us access to their FIBs.

REFERENCES

[1] D. Meyer, L. Zhang, and K. Fall, "Report from the IAB Workshop on Routing and Addressing," RFC 4984, IETF, 2007.
[2] G. Huston, "BGP routing table analysis reports," http://bgp.potaroo.net/.
[3] T. Bu, L. Gao, and D. Towsley, "On characterizing BGP routing table growth," *Comput. Netw.*, vol. 45, no. 1, pp. 45–54, May 2004.
[4] L. Cittadini, W. Muhlbauer, S. Uhlig, R. Bush, P. Francois, and O. Maennel, "Evolution of Internet address space deaggregation: Myths and reality," *IEEE J.Sel. A. Commun.*, vol. 28, no. 8, pp. 1238–1249, 2010.
[5] A. Elmokashfi, A. Kvalbein, and C. Dovrolis, "BGP churn evolution: a perspective from the core," *IEEE/ACM Trans. Netw.*, vol. 20, no. 2, pp. 571–584, 2012.
[6] A. Elmokashfi and A. Dhamdhere, "Revisiting BGP churn growth," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 1, pp. 5–12, Dec. 2013.
[7] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE JSAC*, vol. 17, no. 6, pp. 1083 –1092, 1999.
[8] R. Bolla and R. Bruschi, "RFC 2544 performance evaluation and internal measurements for a Linux based open router," in *IEEE HPSR*, 2006, p. 6.
[9] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving subsecond IGP convergence in large IP networks," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 3, pp. 35–44, 2005.
[10] X. Zhao, D. J. Pacella, and J. Schiller, "Routing scalability: an operator's view," *IEEE JSAC*, vol. 28, no. 8, pp. 1262–1270, 2010.
[11] V. Khare, D. Jen, X. Zhao, Y. Liu, D. Massey, L. Wang, B. Zhang, and L. Zhang, "Evolution towards global routing scalability," *IEEE JSAC*, vol. 28, no. 8, pp. 1363–1375, 2010.
[12] J. Fu and J. Rexford, "Efficient IP-address lookup with a shared forwarding table for multiple virtual routers," in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT '08, 2008, pp. 1–12.
[13] K. Fall, G. Iannaccone, S. Ratnasamy, and P. B. Godfrey, "Routing tables: Is smaller really much better?" in *ACM HotNets-VIII*, 2009.
[14] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: towards entropy bounds and beyond," in *ACM SIGCOMM 2013*, 2013, pp. 111–122.
[15] ——, "Compressing IP forwarding tables: towards entropy bounds and beyond," Technical Report, 2014, available online: http://arxiv.org/abs/1402.1194.
[16] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 1–10, 1998.
[17] I. Ioannidis and A. Grama, "Level compressed DAGs for lookup tables," *Comput. Netw.*, vol. 49, no. 2, pp. 147–160, 2005.
[18] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.
[19] R. Draves, C. King, S. Venkatachary, and B. Zill, "Constructing optimal IP routing tables," in *IEEE INFOCOM*, 1999.
[20] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis, "SMALTA: practical and near-optimal FIB aggregation," in *ACM CoNEXT*, 2011, pp. 1–12.
[21] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *ACM SIGCOMM*, 1997, pp. 25–36.
[22] J. Hasan and T. N. Vijaykumar, "Dynamic pipelining: making IP-lookup truly scalable," in *ACM SIGCOMM*, 2005, pp. 205–216.
[23] A. McAuley and P. Francis, "Fast routing table lookup using CAMs," in *IEEE INFOCOM*, 1993, pp. 1382–1391.
[24] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," in *ACM SIGCOMM*, 2003, pp. 201–212.
[25] P. Gupta, B. Prabhakar, and S. P. Boyd, "Near optimal routing lookups with bounded worst case performance," in *IEEE INFOCOM*, 2000, pp. 1184–1192.
[26] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *ACM SIGCOMM*, 2010, pp. 195–206.
[27] M. Zec, L. Rizzo, and M. Mikuc, "DXR: towards a billion routing lookups per second in software," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 29–36, 2012.
[28] Y. Liu, S. O. Amin, and L. Wang, "Efficient FIB caching using minimal non-overlapping prefixes," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 1, pp. 14–21, Jan. 2012.
[29] W. Wu, *Packet Forwarding Technologies*. Auerbach, 2008.
[30] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley-Interscience, 1991.
[31] I. Ioannidis, "Algorithms and data structures for IP lookups," Ph.D. dissertation, Purdue University, Department of Computer Sciences, 2005.
[32] EZChip, "NP-4: 100-Gigabit Network Processor for Carrier Ethernet Applications," http://www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf, 2011.
[33] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
[34] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and indexing labeled trees, with applications," *J. ACM*, vol. 57, no. 1, pp. 1–33, 2009.
[35] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992.
[36] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Comput. Surv.*, vol. 39, no. 1, 2007.
[37] A. Andersson and S. Nilsson, "Faster searching in tries and quadtrees – An analysis of level compression," in *Algorithms–ESA'94*, ser. Lecture Notes in Computer Science. Springer, 1994, vol. 855, pp. 82–93.
[38] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *ACM SIGCOMM*, 1997, pp. 3–14.
[39] J. Choi, J. H. Park, P. chun Cheng, D. Kim, and L. Zhang, "Understanding BGP next-hop diversity," in *INFOCOM Workshops*, 2011, pp. 846–851.
[40] S. Sahni and K. S. Kim, "Efficient construction of multibit tries for IP lookup," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 650–662, 2003.

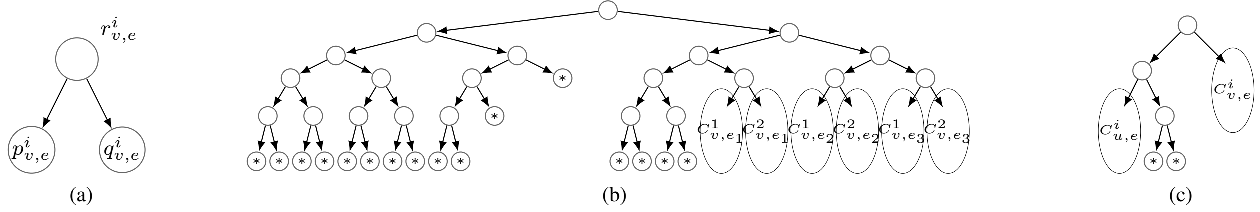


Fig. 7: Illustration for the proof of Theorem 1: (a) link-gadget $C^i_{v,e}$ for node v, incident edge e, and $i \in \{1, 2\}$; (b) node gadget N_v for node v with incident edges e_1, e_2, and e_3; and (c) edge gadget Q^i_e for edge $e = (u, v)$ and $i \in \{1, 2\}$. Asterisks $(*)$ mark unique, unshareable labels.

[41] P. Berman and T. Fujito, "On approximation properties of the independent set problem for low degree graphs," *Theory of Computing Systems*, vol. 32, no. 2, pp. 115–132, 1999.

[42] J. Katajainen and E. Mäkinen, "Tree compression and optimization with applications," *International Journal of Foundations of Computer Science*, vol. 1, no. 4, pp. 425–447, 1990.

[43] V. V. Vazirani, *Approximation algorithms*. Springer-Verlag New York, Inc., 2001.

[44] A. Majdán, "mpa-library," https://github.com/andmaj/mpa-library, 2014.

[45] D. S. Miller, "net_test_tools," https://kernel.googlesource.com/pub/scm/linux/kernel/git/davem/net_test_tools.

[46] P. Hick, kc claffy, and D. Andersen, "CAIDA Anonymized Internet Traces," http://www.caida.org/data/passive.

[47] H. Song, M. S. Kodialam, F. Hao, and T. V. Lakshman, "Scalable IP lookups using Shape Graphs," in *IEEE ICNP*, 2009, pp. 73–82.

[48] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables," in *IEEE INFOCOM*, 2013, pp. 1231–1239.

[49] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *ACM SIGCOMM*, 2014.

[50] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *INFOCOM*, 2013, pp. 2049–2057.

APPENDIX

Proof of Theorem 1: MINLEVCOMPDAG(k) is trivially in NP, as for any given trie (T, l) and a witness (D, f) we can verify in time proportional to the number of nodes in T that $(T, l) \sim (D, f)$ and the number of pointers $M(D, f)$ is at most k. To show that it is also NP-hard, we give a Karp-reduction from the Maximal Independent Set on Cubic Graphs problem (MAXINDSET3, [41]) to MINLEVCOMPDAG.

Definition 3. MAXINDSET3(l): *given a 3-regular graph $G(V, E)$ and an integer l, is there a set $V' \subset V$ with $|V'| \leq l$ so that no two nodes in V' are connected by an edge in E?*

Given a connected 3-regular graph $G(V, E), n = |V|$, and integer l, we construct a proper, binary, leaf-labeled trie (T, l) and an integer k, so that there is an independent set of size l in $G(V, E)$ if and only if MINLEVCOMPDAG on (T, l) can be solved with at most k pointers. Let $E(v)$ denote set of (three) edges incident to node $v \in V$. We use the following gadgets.

- For each $v \in V$, $e = (v, u) \in E$, $i \in \{1, 2\}$, the *link-gadget* $C^i_{v,e}$ is a proper trie of 3 nodes (see Fig. 7a). Let the root be $r^i_{v,e}$ and label the left child with the unique label $p^i_{v,e}$ and the right child with $q^i_{v,e}$. Note that, in our construction, only link-gadgets are shareable in (T, l).
- For each $v \in V$, the *node-gadget* N_v is a binary trie of depth 4, with 10 unshareable leaves on the left and 4 unshareable leaves plus 6 link-gadgets $C^i_{v,e_j} : e_j \in E(v), i \in \{1, 2\}$ on the right (Fig. 7b).
- Finally, for each $e = (u, v) \in E$ and $i \in \{1, 2\}$, the *edge-gadget* Q^i_e is a binary trie containing the link-gadgets $C^i_{u,e}$ at depth 2 and $C^i_{v,e}$ at depth 1 (Fig. 7c).

Collect all gadgets into a labeled forest and connect this into a rooted binary trie (T, l). The details of this construction are uninteresting here, it is enough to know that we can do this so that the following observations (in particular, *(i)*) hold.

Observation 2. *For any $(D, f) \sim (T, l)$, there is a prefix DAG $(D', f') \sim (D, f)$ with $M(D', f') \leq M(D, f)$ so that*

(i) the root of the node- and edge gadgets appear at the first level of (D', f') and this level uses $R = O(n)$ pointers;
(ii) for each $v \in V$, either all $C^i_{v,e_j} : e_j \in E(v), i \in \{1, 2\}$ are shared and $M(N_v) = 30$, or otherwise $M(N_v) = 32$;
(iii) for each $e \in E, i \in \{1, 2\}$, $M(Q^i_e) = 8$; and so
(iv) $\{v \in V : M(N_v) = 30\}$ is an independent set in G.

To see *(ii)*, observe that if all link-gadgets in N_v are shared then we can level-compress the left sub-trie to 16 pointers and the right one to $8+4$ pointers (8 for levels 2–4 contracted and 4 for the uniquely labeled nodes at left, using the convention that whenever a link-gadget is shared we count it to the corresponding edge-gadget) and 2 additional pointers from the root, otherwise we can compress N_v to a single level with 32 pointers. Regarding *(iii)*, if for $Q^i_e : e = (u, v)$ both $C^i_{u,e}$ and $C^i_{v,e}$ are shared then we need to leave Q^i_e as binary (so $M(Q^i_e) = 10$), otherwise we can level-compress the first two levels (which sets $C^i_{u,e}$ shareable) or the last two levels (which sets $C^i_{v,e}$ shareable) or all levels (which sets neither shareable), each option needing 8 pointers. In addition, if Q^i_e holds 10 pointers then both $C^i_{u,e}$ and $C^i_{v,e}$ are shared, removing one of which we could compress Q^i_e down to 8 pointers losing at most 2 pointers at one of the node-gadgets N_u or N_v (by *(ii)*). It follows that the nodes $v \in V$ for which the node-gadget N_v is compressed to 30 pointers in fact make up an independent set in G (item *(iv)*), which concludes the proof as follows.

Corollary 3. *There is an independent set V' in G with $|V'| \geq l$, if and only if there is a prefix DAG $(D, f) \sim (T, l)$ with at most $k = R + \sum_{v \in V} M(N_v) + \sum_{i=1}^{2} \sum_{e \in E} M(Q^i_e) = R + 30l + 32(n - l) + {}^{3n}/_{2}(2 * 8) = R + 56n - 2l$ pointers.* ■