Adaptive Data Structures for IP Lookups

IOANNIS IOANNIDIS, ANANTH GRAMA, and MIKHAIL ATALLAH
Purdue University

The problem of efficient data structures for IP lookups has been well studied in the literature. Techniques such as LC tries and extensible hashing are commonly used. In this paper, we address the problem of generalizing LC tries, based on traces of past lookups, to provide performance guarantees for memory suboptimal structures. As a specific example, if a memory-optimal (LC) trie takes 6 MB and the total memory at the router is 8 MB, how should the trie be modified to make best use of the 2 MB of excess memory? We present a greedy algorithm for this problem and prove that, if for the optimal data structure there are b fewer memory accesses on average for each lookup compared with the original trie, the solution produced by the greedy algorithm will have at least $\frac{9\times b}{11}$ fewer memory accesses on average (compared to the original trie). An efficient implementation of this algorithm presents significant additional challenges. We describe an implementation with a time complexity of $O(\xi(d)n \log n)$ and a space complexity of $O(n)$, where n is the number of nodes of the trie and d its depth. The depth of a trie is fixed for a given version of the Internet protocol and is typically $O(\log n)$. In this case, $\xi(d) = O(\log^2 n)$. We also demonstrate experimentally the performance and scalability of the algorithm on actual routing data.

Categories and Subject Descriptors: E.2

General Terms: Algoriths

Additional Key Words and Phrases: IP lookups, level compression

1. INTRODUCTION AND MOTIVATION

The problem of developing efficient data structures for IP lookups is an important and well-studied one. Given an address, a lookup table returns a unique output port corresponding to the longest matching prefix of the address. Specifically, given a string s and a set of prefixes S, a lookup for s should find the

A preliminary version of this paper appeared in the *Proceedings of INFOCOM* (2003).
This work is supported in part by the Computing Research Institute of Purdue University, National Science Foundation grants ACI-9875899, ACI-9872101, EIA-9903545, IIS-0219560, IIS-0312357, and IIS-0242421, contract N00014-02-1-0364 from the Office of Naval Research, by sponsors of the Center for Education and Research in Information Assurance and Security, and by Purdue Discovery Park's e-Enterprise Center. Computing equipment used for this work was supported by National Science Foundation and by the Intel Corp.
Authors' address: Department of Computer Sciences, Purdue University, 250 N. University Street, West Lafayette, IN, 47907-2066; email: {ioannis,ayg,mja}@cs.purdue.edu.

longest prefix s' in S that is also a prefix of s. The most frequently used data structure to represent a prefix set is a trie because of its simplicity and dynamic nature. A variation that has in recent years gained in popularity is the combination of tries with hash tables. The objective of these techniques is to create local hash tables for the parts of the trie that are most frequently accessed. The obvious obstacle to turning the entire trie into a hash table is that such a table would not fit into the router's memory. The challenge is to identify parts of the trie that can be expanded into hash tables without exceeding available memory, while yielding most benefit in terms of memory accesses.

A scheme combining the benefits of hashing without increasing associated memory requirements, called *level compression*, is described in Nilsson and Karlsson [1998]. This scheme is based on the observation that parts of the trie that are full subtries can be replaced by a hash table of the leaves of the subtrie without increasing the memory needed to represent the trie and without losing any of the information stored in it. This simple, yet powerful, idea reduces the expected number of memory accesses for a lookup to $O(\log\log n)$ or $O(\log^* n)$, depending on the probability distribution of the input. Here n is the size of the original trie. In Waldvogel et al. [1997], an extension of level compression, referred to as *extensible hashing*, was presented. In extensible hashing, certain levels of the trie are filled and subsequently level compressed. These levels are selected to coincide with frequent prefix lengths with the expectation that the trade-off between extra storage space and performance is favorable. A natural extension of the scheme would be to turn into hash tables those parts of a trie that are close to being full and frequently accessed in a systematic fashion. We would like this notion of "close" to vary with the trie, the access characteristics, and the memory constraints.

As a specific example, we are given a set of prefixes with their respective frequencies (probabilities) of access. We are also given a constraint on the total router memory, say, 8 MB. If the trie for the prefixes requires only 6 MB of memory, we would like to build hash tables in the trie to best utilize the 2 MB of excess memory on the router. In general, the problem of building the optimal data structure for a set of prefixes has two parameters. The first is the access statistics of the prefixes, which determines average case lookup time. The second parameter is the memory restriction. Building hash tables in a trie reduces the average lookup time but requires extra memory. The decision to build a hash table for a certain subtrie should depend on the fraction of accesses going through this subtrie and the memory requirement of this modification.

We can formulate a generalization of the level compression and extensible hashing schemes as a variation of the knapsack problem. The items to be included in the knapsack are subtries. The gain of an item is the reduction in average lookup time that results from level compressing this subtrie, and its cost is a function of the number of missing leaves in the subtrie (in other words, the memory overhead of compressing the subtrie). The key difference between this variation and a traditional knapsack is that items are *not static*, rather, their attributes vary during the process of filling the knapsack. The correspondence between the parameters of this formulation and the parameters of the table lookup problem is very natural and can be defined precisely in a straightforward manner. An advantage of this formulation is that there is no shortage of approximation schemes for knapsack. In fact there is a hierarchy of approximation algorithms, starting with the extremely fast greedy algorithm, having an approximation ratio of two, to elaborate polynomial time approximation schemes. For a comprehensive study of the knapsack problem and its variations see Martello and Toth [1990].

We would like to note that, even though the motivation for this work has been IP routing, it has applicability in a variety of domains, such as information retrieval and index structures in databases, that require longest prefix matching. The proposed abstraction of the routing table as a trie does not carry any restrictions specific to the problem.

2. OVERVIEW OF THE ALGORITHM AND RELATED RESEARCH

We first describe a greedy algorithm for level compressing different parts of a trie according to their access rates and storage requirements. The algorithm resembles the known greedy approximation algorithm for knapsack. A subtrie is selected on the basis of the ratio of the decrease in the average lookup time resulting from its level compression and the required memory of the corresponding hash table. The process continues until no other item can be added to the knapsack. Although greedy algorithms are known for their simplicity, in this case there is a peculiarity: the attributes of the items are not static but vary over the execution of the algorithm. Specifically, selecting a subtrie for level compression has a cascading effect on items on the paths to the root and the leaves. This complicates the algorithm, analytical performance bounds, and implementation significantly. We show that the greedy algorithm can approximate the optimal solution within a factor of $\frac{9}{11}$. The approximation is with respect to the original trie. In other words, if for the optimal data structure, there are b fewer memory accesses on average for each lookup compared with the unprocessed trie, the solution produced by the greedy algorithm will have at least $\frac{9\times b}{11}$ fewer memory accesses on average. We note that the problem is known to be NP-complete [Cheung et al. 1999].

We also describe data structures needed for an efficient implementation of the algorithm. Since a router needs to invoke the algorithm often, even a quadratic dependence on the size of the trie would severely restrict scheme's usefulness. We describe an implementation with a time complexity of $O(\xi(d)n \log n)$ and a space complexity of $O(n)$, where n is the number of nodes of the trie and d its depth. The depth of a trie is fixed for a given version of the Internet protocol and is typically $O(\log n)$. In this case, $\xi(d) = O(\log^2 n)$.

Finally, we present experimental evidence that the proposed algorithm consistently yields better data structures (in terms of average-case lookup cost) than extensible hashing. Evidence relating to scalability issues is presented as well.

2.1 Related Research

The literature on efficient implementation of IP routing is impressively varied. The classical implementation of IP routing for the BSD kernel is described in Sklower [1991]. True to the spirit of UNIX, simplicity is not sacrificed for performance. In Gupta et al. [1998], McAuley et al. [1995], Newman et al. [1997], and Narlikar and Zane [1991], hardware and cache-based solutions are proposed. Hardware solutions tend to become expensive and outdated, while cache-based solutions do not avoid the central issue of prefix matching. A similar argument can be made in the case of Sikka and Varghese [2000], where lookups are accelerated using memory placement and pipelining. Some protocol-based solutions have emerged [Bremler-Barr et al. 1999; Chandranmenon and Varghese 1996; Labovitz et al. 1997; Newman et al. 1997; Parulkar et al. 1995; Rekhter et al. 1997], but all of these demand modifications to the current Internet protocol and raise the complexity of routing without completely avoiding the prefix matching problem.

Recent research has focused on algorithmic solutions [Crescenzi et al. 1999; Degermark et al. 1997; Lampson et al. 1998; Srinivasan and Varghese 1998; Waldvogel et al. 1997]. The advantage of these is their transparency to protocol and to advances in hardware platforms. In Nilsson and Karlsson [1998], the original level compression scheme is described. A probabilistic analysis of the performance of level compression is presented in Andersson and Nilsson [1994]. In Srinivasan and Varghese [1998], a scheme called controlled prefix expansion is presented for solving the generalized level compression problem. The restriction in this setting is the average number of memory accesses and the optimization target the memory requirements. The only effort we are aware of on formulating a generalization of level compression to include memory constraints and providing a solution is presented by Cheung and McCanne [1999]. Their solution is a simple, dynamic-programming, pseudopolynomial time algorithm. An approximation using Lagrange multipliers is also described, although no constant bound on the error is derived for this approximation scheme. They formalize memory constraints in the form of an arbitrary memory hierarchy. Cheung et al. [1999] also show that the problem is NP-complete even for one-level memory.

3. THE ALGORITHM

We formulate the generalized level compression problem as a variation of the knapsack problem. The main difference between this formulation and the classical knapsack is the dynamic nature of the items. In knapsack, selecting one item does not alter the attributes of the other items. This is not true in our case because subtries necessarily overlap and contain each other. Selecting an item, or in other words, level compressing a subtrie, will level compress some other subtries, making them irrelevant for the rest of the execution, while it will modify the gain and cost of those items corresponding to overlapping subtries.

These dependencies are not arbitrary. They follow from the hierarchical nature of the trie structure. We can use this property to achieve a constant approximation bound and reduce the runtime and space complexities. We first formulate the problem by defining what an item is and how we calculate its initial attributes. We then describe how selecting an item affects the attributes of other items. Finally, we describe a greedy algorithm, which works along these lines and derive an efficient implementation.

3.1 Definitions

In the following, we assume that for each leaf of the trie we have access statistics available to us. In other words, for each path from the root to a leaf, we know what is the probability that the next lookup will access this path. The probability distribution is assumed to be static. We also consider that the root of a trie is at level 0, its children at level 1, and so on. Finally, the depth of a node k is denoted by $depth(k)$ and is the length of the path from k to the root of the trie.

We want the trie structure to reflect the expected number of accesses going through a node. We formalize the notion of a trie with access statistics for the addresses. We call this structure a weighted trie. Each node is assigned a weight. In our context, the weight of a leaf is the probability that the path ending at that leaf is looked up, although it is not necessary for the formal definition. If an internal node has two children, its weight should be equal to the sum of the weights, as we assume that the longest matching prefix is returned. An internal node can be the answer to such a query only if it has only one child, in which case the weight of the internal node cannot be less than the weight of the child. The weight of a node is not the access probability, but it can be normalized to be so.

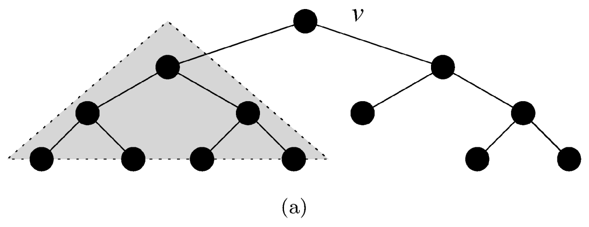
Definition 3.1. A *weighted trie* is a pair (T, w), where T is a trie and w a function that maps the nodes of T to the set of positive reals with the property that $w(v) = w(v_1) + w(v_2)$, if v has two children v_1, v_2, and $w(v) \geq w(v_1)$, if v has one child v_1.

During the level compression process, each level-compressed subtrie can be identified by its root and its depth. We call a subtrie candidate for level compression, an *item*. Adding an item i to the knapsack corresponds to a decision to level compress the subtrie corresponding to i. Items, like subtries, are identified by their root and depth.

Definition 3.2. An *item* is a pair (v, k), where v is a node of the trie and k a positive integer. We say that k is the depth of the item and v the root. If $i = (v, k)$, $root(i) = v$ and $depth(i) = k$.

Selecting an item (v, k) for level compression creates a hash table from the subtrie rooted at v and having depth k. This requires a certain amount of extra memory, expressed as the number of trie nodes we need to fill. This quantity is not fixed but depends on the sequence of items that have been previously selected. We denote a sequence of j items already selected for level compression by p^j. We will use the function $capacity(v, k, p^j)$ to denote the reduction in the knapsack capacity resulting from selecting (v, k) as the $(j + 1)$st item in the sequence p^{j+1}. For simplicity, we use j instead of p^j whenever the sequence is either known or unimportant. We show a simple case of the capacity of an item varying due to different items having been selected before in Figure 1.

We first define $capacity(v, k, 0)$ for the initial items. The values of *capacity* when items have been added to the knapsack are defined implicitly by the update rules described in the next section. When a subtrie is level compressed,



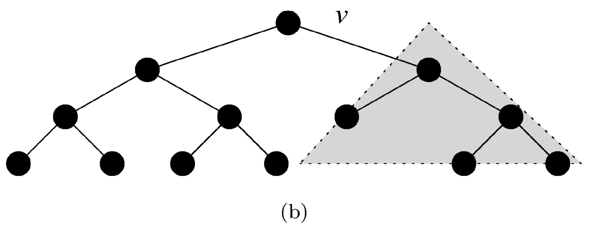


Fig. 1. Item $(v, 3)$ has different capacities in (a) and (b). (a) Selecting the left subtrie leaves the capacity of $(v, 3)$ the same. (b) Selecting the right subtree makes the capacity of $(v, 3)$ zero.

the internal nodes become irrelevant and only the leaves carry information. Therefore, the hash table is a table of only the leaves and the extra memory such a table requires is equal to the number of missing leaves for that subtrie. This is a slightly simplified definition of space requirements for level compressing a subtrie. We discuss a more precise definition for capacity and show that it does not affect the results in this paper in a later section.

Definition 3.3. For every $i = (v, k)$, $capacity(v, k, 0)$ is the number of missing nodes from the level of depth k of the subtrie rooted at v.

We assign to each item i a benefit corresponding to the reduction in the number of accesses due to level compressing the subtrie corresponding to i. Similar to the capacity attribute, benefit depends on p^j. We define the initial capacities first and the rest of the values through the update rules. Obviously, level compressing a subtrie of depth one does not result in fewer memory accesses. Only nodes at depth greater than 1 benefit from level compression. Naturally, the reduction of the overall average search time resulting from level compressing a subtrie is proportional to the expected number of accesses in this subtrie. In the following definition, we sum the weights of the nodes starting from the second level of a subtrie because level compressing a subtrie of depth one does not reduce the lookup time. Two memory accesses have to be performed anyway to the hash table before the desired entry is reached [Nilsson and Karlsson 1998]. First the head of the table has to be accessed, before it is decided that it is indeed a hash table. Then, the target entry has to be accessed. A more sophisticated mechanism, where the repeated access is avoided, if the first entry is the target, introduces an extra condition check, which results in overall slower lookups. Since we count only main memory accesses, wo do not consider the larger issue of caching in the presence of level compression. In any case, the definition can be modified to add to the benefit the weight of the left child of the root of the item, without altering the proofs in any significant way.

Definition 3.4. For every $i = (v, k)$,

$$benefit(v, k, 0) = \sum_{l=2}^{k} \sum_{u \in C(v,l)} w(u),$$

where $C(v, l)$ is the set of the descendants of v that are at depth $(depth(v) + l)$.

We want to order items in such a way that the next item to be added to the knapsack is the maximum element according to this order. Intuitively, $density(v, k, i)$ corresponds to the ratio of $benefit(v, k, i)$ and $capacity(v, k, i)$. However, in the initial phase of the algorithm, there will be items for which capacity is zero. These correspond to full subtries, which can be level compressed without reducing the capacity of the knapsack. In this case, the ratio is not defined. Also, a rule must be provided to resolve the ties between items with equal densities. For these reasons, we formally define *density* as a partially ordered function mapping the set of items to a set of cardinality $(nd)^2$. Here, n is the number of nodes of the trie and d its depth. The *density* function must satisfy the following: $density(v_1, k_1, i_1) < density(v_2, k_2, i_2)$, where $(v_1, k_1) \neq (v_2, k_2)$, if and only if one of the following holds:

(1) $benefit(v_1, k_1, i_1) \times capacity(v_2, k_2, i_2) <$
$benefit(v_2, k_2, i_2) \times capacity(v_1, k_1, i_1)$.
This rule corresponds to the case where the density ratio can be defined for both items and there is no tie, or it can be defined for only one item. In the latter case, the item with zero capacity takes priority.

(2) $benefit(v_1, k_1, i_1) \times capacity(v_2, k_2, i_2) =$
$benefit(v_2, k_2, i_2) \times capacity(v_1, k_1, i_1)$ and
$capacity(v_1, k_1, i_1) > capacity(v_2, k_2, i_2)$.
This rule corresponds to the case where the density ratio can be defined for both items and there is a tie. The item with smaller capacity takes priority.

(3) $benefit(v_1, k_1, i_1) \times capacity(v_2, k_2, i_2) =$
$benefit(v_2, k_2, i_2) \times capacity(v_1, k_1, i_1)$,
$capacity(v_1, k_1, i_1) = capacity(v_2, k_2, i_2)$ and
$depth(v_1) > depth(v_2)$.
This rule corresponds to the case in which either the density ratio can be defined for both items and there is a tie for both capacity and benefit or the density ratio can be defined for neither of them. The item that is higher in the trie takes priority. The motivation for this tiebreaker is the top-down nature of the level compression process. If the two items are located in unrelated parts of the trie, it does not make a difference which one is picked first. However, if the root of one item is ancestor of the root of the other, we pick the one that will be level compressed first.

(4) $capacity(v_1, k_1, i_1) = capacity(v_2, k_2, i_2)$,
$benefit(v_1, k_1, i_1) \times capacity(v_2, k_2, i_2) =$
$benefit(v_2, k_2, i_2) \times capacity(v_1, k_1, i_1)$,
$depth(v_1) = depth(v_2)$ and $v_1 < v_2$.
If the previous tiebreaker cannot be applied and the two items have different roots, we can pick either one first. Assuming there is a linear ordering of

the nodes (e.g., a depth-first ordering, although any ordering suffices), we can use this ordering to resolve the tie.

(5) $v_1 = v_2$, $\mathit{capacity}(v_1, k_1, i_1) = \mathit{capacity}(v_1, k_2, i_2) = 0$ and $k_1 > k_2$.
At first look, this rule appears counterintuitive. If two items have the same root and zero capacities, they correspond to full or already filled subtries and the deeper trie has a larger benefit and should be picked. The reason we give priority to the smaller subtrie is that since the capacity of both subtries is zero, they will both be added to the knapsack eventually. In the following proofs, there is the implicit notion that to add an item (v, k) to the knapsack, all the (v, k') items, $k' < k$, must have already been added. This artificial rule preserves this notion without contradicting the level compression process.

In the rest of the paper, for the sake of simplicity, we also denote the attributes of an item $i = (v, k)$ at round j as *benefit*(i, j), *capacity*(i, j), and *density*(i, j), instead of using the explicit notation.

Remark. An important property of items is that their densities follow the hierarchical order of the trie. Let (v, k) and $(v, k + 1)$ be two items having the same root and differing only at one level. It is easy to see that $\mathit{density}(v, k, 0) \geq \mathit{density}(v, k+1, 0)$. The reason is that $\mathit{capacity}(v, k+1, 0) \geq 2 \cdot \mathit{capacity}(v, k, 0)$, but $\mathit{benefit}(v, k + 1, 0) \leq 2 \cdot \mathit{benefit}(v, k, 0)$. We will later prove that a slightly modified version of this property holds throughout the execution of the greedy algorithm, for similar reasons.

3.2 The Update Rules

We describe the rules by which $\mathit{capacity}(v, k, i)$ and $\mathit{benefit}(v, k, i)$ are obtained from $\mathit{capacity}(v, k, i - 1)$ and $\mathit{benefit}(v, k, i - 1)$, respectively, for all $i > 0$. We assume that the ith item added to the knapsack is (v', k'). We must first define when two items interact with each other. Informally, an item i interacts with i' when the corresponding subtries share more than one node. If they share only one node, this must be the root of the lower placed subtrie, which can be level compressed as it is after the higher placed subtrie has been level compressed. We define, formally, when level compressing one item affects another as the *overlaps* relation. A special case of interaction is when an item is entirely contained in another.

Definition 3.5. We say that item (v_1, k_1) *overlaps* with item (v_2, k_2), if v_2 is a descendant of v_1 and $\mathit{depth}(v_1) + k_1 - 1 \geq \mathit{depth}(v_2)$, or if v_1 is a descendant of v_2 and $\mathit{depth}(v_2) + k_2 - 1 \geq \mathit{depth}(v_1)$. Furthermore, if $\mathit{depth}(v_1) + k_1 \geq \mathit{depth}(v_2) + k_2$, we say that (v_1, k_1) contains (v_2, k_2) and we write $(v_2, k_2) \in (v_1, k_1)$.

Remark. There are two ways of viewing the effect of selecting an item i on an overlapping item i'. One is to remove i' from the set of candidate items. The second is to replace it with a collection of the items contained in i'. In the rest of the paper we choose the second view. However, for implementation purposes it is better to choose the first, since it results in more efficient code.

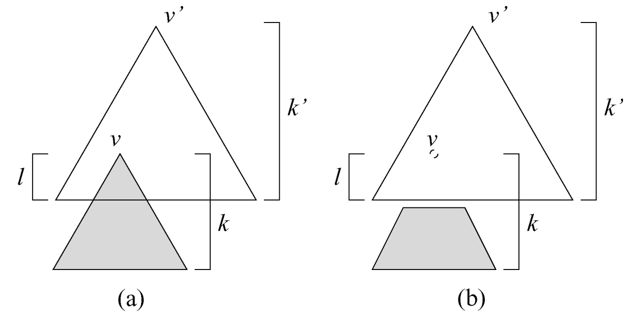


Fig. 2. Updating overlaps: (a) before updating (v, k); (b) after updating.

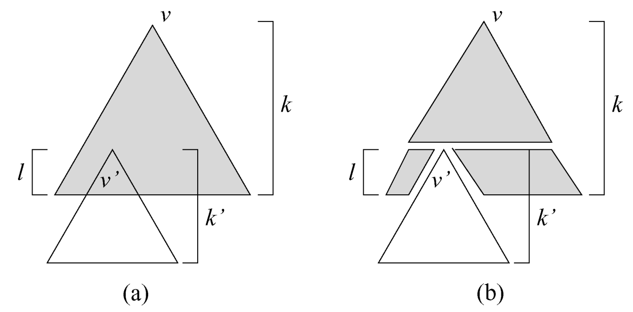


Fig. 3. Updating overlaps: (a) before updating (v, k); (b) after updating.

If two items overlap, adding one of them to the knapsack alters the attributes of the other. After adding an item we need to update the benefit and capacity of all items overlapping with it. For the following list of update rules, assume that (v, k) overlaps with (v', k'). If they do not, $\mathit{benefit}(v, k, i) = \mathit{benefit}(v, k, i-1)$ and $\mathit{capacity}(v, k, i) = \mathit{capacity}(v, k, i-1)$.

—Update rule 1: $\mathit{depth}(v) > \mathit{depth}(v')$ (Figure 2). If $(v, k) \in (v', k')$, $\mathit{capacity}(v, k, i) = \mathit{benefit}(v, k, i) = 0$. This is because level compressing a subtrie level compresses all the subtries included in it. Otherwise, let $l = \mathit{depth}(v') + k' - \mathit{depth}(v)$, the height of v in (v', k'). We want (v, k), after selecting (v', k'), to be the sum of all items contained in (v, k) and not overlapping with (v', k'). Due to the top-down nature of level compression, this is exactly the effect (v, k) would have on the solution if added as the $(i+1)$st item of the knapsack. Therefore, $\mathit{benefit}(v, k, i) = \sum_{u \in C(v,l)} \mathit{benefit}(u, k-l, i)$. We note that all items whose root is in $C(v, l)$ do not overlap with (v', k') and the sum is well defined. Similarly, $\mathit{capacity}(v, k, i) = \sum_{u \in C(v,l)} \mathit{capacity}(u, k-l, i)$.

—Update rule 2: $v = v'$. This is a case of one item extending another. If $k < k'$, $(v, k) \in (v', k')$ and $\mathit{capacity}(v, k, i) = \mathit{benefit}(v, k, i) = 0$. Otherwise, $(v', k') \in (v, k)$ and $\mathit{capacity}(v, k, i) = \mathit{capacity}(v, k, i-1) - \mathit{capacity}(v', k', i-1)$ and $\mathit{benefit}(v, k, i) = \mathit{benefit}(v, k, i-1) - \mathit{benefit}(v', k', i-1)$.

—Update rule 3: $\mathit{depth}(v) < \mathit{depth}(v')$ (Figure 3). Let $l = \mathit{depth}(v) + k - \mathit{depth}(v')$. We have

$$\mathit{benefit}(v, k, i) = \mathit{benefit}(v, k, i-1) - \mathit{benefit}(v', l, i-1)$$

and

$$capacity(v,k,i) = \sum_{u \in C(v',k-l)-\{v'\}} capacity(u,l,i)$$

$$+ capacity(v',k-l,0) \cdot 2^{l} - capacity(v',l,0) \cdot 2^{k'-l}.$$

The reason the $capacity(v',l,0) \cdot 2^{k'-l}$ term must be subtracted from the capacity is that, due to the top-down manner in which level compression is applied, (v,k) is level compressed first. At this point (v',k') breaks into a collection of subtries rooted at level $depth(v')+l$. However, while adding (v',k') to the knapsack, each missing node at this level that is a descendant of v' has incurred a cost as large as the number of leaves in the subtrie that must be filled in (v',k') due to this missing node. These subtries do not have to be filled any more, so their cost must be removed. It appears that the capacity of an item could become negative in this case. We will see that this is impossible, if we pick items in a greedy fashion. The term $capacity(v',k-l,0) \cdot 2^{l}$ has to be added for a similar reason. The missing nodes at depth $k-l$ from v will generate 2^{l} missing nodes each at depth k from v.

4. A GREEDY ALGORITHM

The above rules imply a greedy algorithm, which we will call A. In each step, the item with the highest density is added to the knapsack. Other items are updated according to the update rules. The process continues until the next item to be added exceeds the available knapsack capacity. Algorithm A produces the optimal solution whenever the knapsack is filled to capacity.

THEOREM 4.1. *If algorithm A fills knapsack K to capacity, the solution induced is optimal.*

PROOF. See appendix. □

4.1 Approximation Ratio for Partially Filled Knapsack

In this section we consider only tries whose nodes have either two or no children. These tries cannot be path compressed. We will prove that for such tries the benefit of the solution cannot be less than $\frac{9}{11}$ of that of the optimal solution. We will address some issues arising from combining level compression and path compression later and from alternate definitions of benefit and capacity later.

THEOREM 4.2. *Let (T,w) be a weighted trie whose nodes have either two or no children. Let b be the benefit of the solution produced by A for (T,w) when C extra nodes can be added to the trie and b^* the benefit of the optimal solution for T with capacity C. Then, $\frac{b}{b^*} \geq \frac{9}{11}$.*

PROOF. See appendix. □

4.2 Alternate Benefit and Capacity Definitions

In the previous sections, we have adopted a model of level compression that does not take into account path compression. In simple level compression, this is not a problem, since a subtrie that contains a path that can be path compressed is not going to be full and it is not going to be level compressed. One can apply path compression and level compression in any order and achieve the same results. However, in the case of generalized level compression, it is possible that a path that would be eliminated by path compression is, instead, part of a level-compressed trie. As a consequence, we first need to level compress a trie and then path compress the resulting structure. Of course, some care is needed in the definition of item benefit. The way benefit has been defined in Sections 3.1 and 3.2, a path that would be eliminated by path compression, contributes to the benefit of the items it is part of anyway. One solution is to add the weight of a node in the benefit of an item only if this node has a sibling and to subtract the appropriate weight from any item that breaks a path that would have been compressed into two. The new definition represents the effect of path compression correctly, but it complicates the description of benefit and the subsequent proofs.

Another solution is to give a second child to each node with only one. Suppose the path $\langle v_1, v_2, \ldots, v_i\rangle$ can be path compressed to $\langle v_1, v_i\rangle$. We can transform the original trie by adding a child to every v_j, $2 \leq j < i$. Let the new child of node v_j be v'_j. To incorporate path compression without modifying the benefit definitions, $w(v'_2)$ must be $-2 \cdot w(v_i)$ and $w(v'_j)$ must be $w(v_i)$ for $2 < j \leq i-1$. Finally, the weight for $v'_{(i-1)}$ must be 0, unless $i = 3$, for which $v'_{(i-1)} = v'_2$. In this case we must double the weight of v_3, while having $w(v'_2) = -2 \cdot w(v_3)$, with $w(v_3)$, being the original weight of v_3. The above do not violate the definition of the weighted trie given in Section 3.1. Let an item contain path $\langle v_1, v_2, \ldots, v_k\rangle$, $1 < k < i$. This item also contains nodes $v'_2, \ldots, v'_{(k-1)}$ for a total reduction in its benefit of $(k-1) \cdot w(v_i)$. If $k = i$, the total reduction is $(i-2) \cdot w(v_i)$. In both cases, the adjusted benefit correctly reflects path compression. Some items can have negative benefits, but this requires a minimal change in the ordering properties of the *density* function and no change in the proof of Theorem 4.1. Alternately, all items that have negative benefit in some round can be removed from the item set, just like all items whose initial capacity is larger than that of the knapsack.

In the above solution to the path compression problem, each item that is affected will have a reduced capacity of one node. However, there is a more serious problem with the definition of capacity. In the original definition, the internal nodes of a subtrie do not affect its space requirements at all. Nevertheless, they should, since level compressing a subtrie saves us the space that normally would be used for the internal nodes. The solution is to change the definition of the initial capacities, so that the number of the internal nodes is subtracted. The result is that some items can start with a negative capacity, and there will be a longer initial phase where items are added to the knapsack without consuming any of its capacity. The question is whether the proofs we have presented work with these modified definitions of benefit and capacity, which are closer to the actual level compression process. The proof of Theorem 4.1 works almost without any change. In fact, the proof works with any definition of benefit and capacity, as long as the initial benefit of an item (v, k) is not larger than a linear function of its depth

Table I. Pseudocode for Algorithm A

```
Input: Weighted trie $(T, w)$, knapsack capacity $K$
Output: Knapsack $S$
Initialize item search tree $I$, ordered by *density*
For = every $v \in T$
    Create $(v, 2)$
    Insert $(v, 2)$ in $I$
$S = \emptyset$
While $I$ is not empty
    $i$ = maximum in $I = (v, k)$
    Remove $i$ from $I$
    If *capacity*$(i) > K$, return $S$
    $S = S \cup \{i\}$
    $K = K -$ *capacity*$(i)$
    Create $(v, k + 1)$
    If $(v, k + 1)$ exists, insert $(v, k + 1)$ in $I$
    For = every $u$ descendant or ancestor of $v$
        $j = (u, l)$ is in $I$
        Remove $j$ from $I$
        If $j$ overlaps with $i$, update $j$
        Insert $j$ in $I$
Return $S$
```

and the initial capacity of (v, k) is no less than an exponential function of its depth.

The proof for Theorem 4.2 must change more significantly to deal with the changes. However, the basis of the proof remains the same. The difference is that the four parts of the trie used in the proof must be a little deeper to accommodate for the fact that even incomplete tries can be level compressed without extra space. As a result, the ratio of the benefit b of the solution returned by the greedy algorithm to the benefit b' of the next item to be added in the knapsack is larger and the approximation ratio $\frac{b}{b+b'}$ will be larger. Therefore, the approximation ratio of Theorem 4.2 can be seen as the lower bound of the approximation ratios for problems whose benefit and capacity definitions have the aforementioned properties and tries whose nodes have two or no children. We have chosen to work with the simpler definitions because they lead to simpler proofs without obscuring the application of greedy algorithms for the generalized level compression problem. Using the transformation described, we can extend Theorem 4.2 to all tries.

COROLLARY 4.3. *Let (T, w) be a weighted trie. Let b be the benefit of the solution produced by A for (T, w) when C extra nodes can be added to the trie and b^* the benefit of the optimal solution for T with capacity C. Then, $\frac{b}{b^*} \geq \frac{9}{11}$.*

5. ALGORITHMIC COMPLEXITY

A naive implementation of the algorithm (see Table I) yields a runtime of $O(n^2d^2)$ and uses $O(nd)$ space, where n is the number of nodes in the trie and d the depth of the trie. For each node $O(d)$ items must be created. At each step, the maximum item is selected scanning all the items in $O(nd)$ time. Each update operation would take $O(n)$ time, and it can be proved there are $O(d)$ such amortized operations at each step. There are $O(nd)$ steps, hence the running time. A more efficient implementation uses only linear space and has a running time of $O(nd^2 \log n)$.

For each node, we create only the item of depth two rooted at that node. The intuition is that since the density of (v, k) is always larger than that of $(v, k+1)$, choosing the latter will always follow picking the former. Each time we choose (v, k), we remove it from the item space and replace it with $(v, k+1)$. This way we only use linear space. We build a search tree on the initial item set. This can be accomplished in $O(n)$ time. Picking the maximum element, and deleting and inserting items in this tree takes time $O(\log n)$.

At each step we need to do the following: find the maximum element (v, k), delete it from the search tree, update all items overlapping with it, and insert $(v, k+1)$, if it exists. Finding, deleting the maximum element, and inserting its successor take time $O(\log n)$, as mentioned. Updating the overlapping items might take $O(n^2 d^2)$ time, if not done carefully.

There are $O(d)$ items overlapping with (v, k) whose root has depth less than that of v. Only items rooted on the path from v to the root of the trie fall in this category. We split the update operations into two categories. The first is the one involving the items mentioned above. The second involves items having v as an ancestor of their root. The first category produces $O(d)$ updates at each step. There are $O(nd)$ steps, for a total of $O(nd^2)$ updates. For the second, a node can be involved in such an update $O(d^2)$ times. It has $O(d)$ ancestors. Each ancestor can involve the node in as many update operations as the number of items rooted at it that can be picked during the execution of the algorithm. Therefore, over the execution of the algorithm, $O(nd^2)$ updates of the second category can be executed. In all, there are $O(nd^2)$ updates.

Each update can be completed in $O(\log n)$ time. Suppose that the overlapping items are (v, k) and (v', k'), with (v, k) the item picked in the current round, and $depth(v) < depth(v')$. To update (v', k'), we need to spend constant time on each node in the set $S = C(v, k) \cap C(v', depth(v) + k - depth(v') + 1)$. Because items expand one level at a time, we need to remove the benefit and capacity resulting from expanding $(v, k-1)$ to (v, k). A detailed description of this process is tedious, since it mainly consists of dealing with special cases. It suffices to say that for each node in S we must subtract its weight from (v', k'). Also, for each node missing from S, but which would be in S had the trie been complete, we need to subtract as much capacity from (v', k') as the hole created in (v', k'). The size of S is $O(n)$, and accessing each node in S gives a running time of $O(n)$ for each update. The case in which $depth(v) \geq depth(v')$ can be treated similarly.

It is possible to reduce the time for an update to $O(\log n)$ by keeping some extra information on each node of the trie. For each node v, we keep a pointer to its left and right neighbors on the level of the trie v is located at. At each node we keep the sum of the weights of all the nodes that are to its left at that level, including itself. We also keep the number of missing nodes to its left at that level. This information can be built from the original trie by a breadth-first traversal. We need to spend $O(1)$ time for building this information on each node, and it takes constant space for each node. We also build two binary search

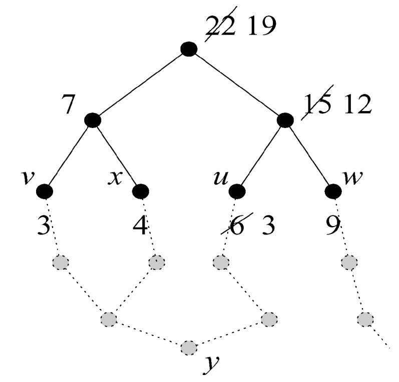


Fig. 4. An auxiliary tree for capacity.

trees on each level of the trie. The leaves of both trees are the nodes of the trie belonging to the corresponding level, ordered from left to right. These trees are similar to an interval tree. The first has to do with weights and the second with the number of missing nodes. Let w be an internal node of the first tree and x, y the leftmost and rightmost leaves, respectively, of the subtree rooted at w. In w, we store the sum of the weights of leaves between x, y, inclusive. It is easy to see that the total space needed for trees of this kind is $O(n)$, and it takes $O(n)$ time to build them. The second tree has similar properties for the number of missing nodes. An illustration of a search tree for the number of missing nodes is given in Figure 4. The structure above nodes v, x, w, and u is the search tree, while the structure below is the trie. All four nodes are at the same depth from the root of the trie. To the left of v, there are three trie nodes missing, compared with a full trie. Between v and x there is one node missing, between x and u, two nodes and between x and w, three nodes. Suppose item $(y, 3)$, where y is the lowest common ancestor of v and u in the trie, is added to the knapsack. This means that any item containing $(y, 3)$ and whose last level is the same as that of nodes v and u should not be accounted for the three missing nodes between v and u. This is done by subtracting three from the stored value of each node of the search tree on the path from the root of the search tree to u. We can find the capacity of any item by accessing the two paths of the appropriate search tree leading to the bottom left and bottom right node of that item. If we want to find the capacity of an item whose bottom left node is v and its bottom right node is w, we need to find their lowest common ancestor in the search tree and from its value subtract the value of its left child $(19 - 7 = 12)$. From this result, we need to subtract the result of subtracting the left child from the next node on the path to w $(12 - (12 - 3) = 3)$ for every node on the path, except w. The result—in this case three—is the number of nodes missing between w and v, not already covered by items in the knapsack. Note that once $(y, 3)$ has been added to the knapsack, x cannot be a bottom corner of any item and its information in the search tree, along with the information of any node inside the paths from v and u to their lowest common ancestor, will never be accessed again. The search tree for weights functions in a similar manner. The space

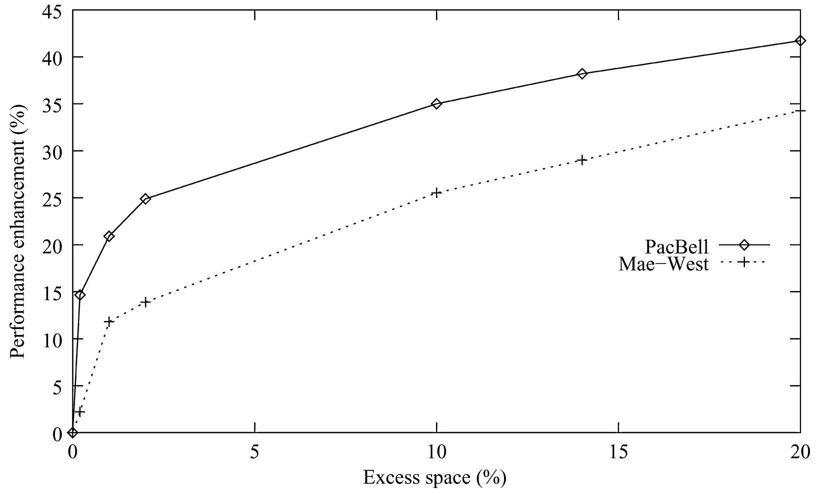


Fig. 5. Performance of the algorithm in terms of memory accesses saved for the Mae-West and PacBell data sets.

requirements for all search trees is $O(n)$, and the time to build them is also $O(n)$. Retrieving information for an item can be done in $O(\log n)$ time.

COROLLARY 5.1. *The knapsack algorithm runs in $O(nd^2 \log n)$ time and $O(n)$ space.*

6. EXPERIMENTAL RESULTS

In addition to the theoretical analysis of the algorithm, we have run a series of experiments to measure its performance and scalability on actual routing data. For this purpose we have used two sets of data: a large Mae-West routing table and a smaller PacBell one. The size of the resulting trie for the Mae-West table was a little over 5×10^5 nodes and is more representative of actual BGP routing tables. The size of the resulting trie for the PacBell table was 33,581 nodes. We used the PacBell data set to ensure that the results for the Mae-West table are representative of the behavior of the algorithm. We did not use the PacBell data set for measuring the running time, since the measured figures were too close to provide any valid insight and we suspect they depended more on external variables, such as the system load, than the input to the code.

For the Mae-West data set, besides the improvement in the expected lookup time, we measured the running time relative to the routing table size and relative to the parameter of extra space. Finally, we applied extensible hashing at levels 16 and 24 of the trie and compared its performance to that of our algorithm. All experiments were run on 2 GHz P4 with 512 MB RAM. The code was written in C++ and executed on a Unix platform. The code was 2,606 lines long. All the code used in these experiments was written by the authors, including code for level compression and extensible hashing.

In Figure 5, we illustrate graphically the performance of the algorithm for the two data sets. In Table II, we show the improvements in performance for the two data sets measured for a number of different excess space sizes. As

Table II. Experimental Measurement of Improvement in Number of Lookups (as a Percentage of That of Level Compression) with Increasing Amounts of Extra Space

	Performance Enhancement (%)	
Extra Memory(%)	Mae-West	PacBell
0.2	2.21	14.66
1	11.8	20.91
2	13.9	24.88
10	25.52	35
14	29.01	38.21
20	34.27	41.73

Table III. Experimental Measurements of Improvement in Number of Lookups (as a Percentage of Those of a Level Compressed Trie) with Increasing Amounts of Extra Memory for Generalized Level Compression and Extensible Hashing for the Mae-West Data Set

Extra Space		Performance Enhancement	
Nodes	%	Level Compression (%)	Extensible Hashing (%)
1×10^3	0.2	2.21	—
5×10^3	1	11.8	—
10×10^3	2	13.9	—
50×10^3	10	25.52	16.92
70×10^3	14	29.01	—
100×10^3	20	34.27	17

Memory is expressed as number of nodes and as percentage of the size of the original trie. Comparison was not possible for all values of extra memory.

we have mentioned, the PacBell data set was used mainly as a control set, to verify that the behavior of our algorithm for the more representative Mae-West data set is qualitatively correct. We can see that the two curves are roughly of the same shape, although the one for the PacBell routing table is consistently higher. In both cases, the performance of the algorithm appears to saturate and it appears doubtful that performance for more than 20% excess space improves significantly.

Table III presents results of the performance enhancement for the Mae-West data set as a percentage of that of simple level compression for varying amounts of excess memory. We note that extra memory is measured as the number of nodes that can be filled. As expected from our theoretical analysis, we observe the benefit of using extra space declines for increasing values of the latter. It is remarkable that for an increase of 10% in total memory, we achieve an increase of 25% in performance.

Table IV presents the runtime of the algorithm for the Mae-West data set and for subsets of various sizes of the routing table, with no extra space allowed. The subsets are created by randomly sampling the prefix set of the original routing table, until a certain percentage of the storage requirements of the original trie is reached. As a consequence, the depth of the resulting trie is kept constant in all cases. This measurement is necessary since a large part of the runtime is consistently spent on precomputation and converting the original trie into a level-compressed one. We note that the runtime for a table of size 50% of the

Table IV. Experimental Measurements (in s) of the Runtime for Routing Tables of Different Sizes

Routing Table Size	Running Time
100	53.5
90	53.4
80	53.35
70	53.37
60	53.31
50	53.28

The first column indicates the size of the table as a percentage of the size of the full routing table. All tables are subsets of the Mae-West routing table.

Table V. Experimental Measurements (in s) of the Runtime for Different Amounts of Extra Space Allowed

Extra Space		
Nodes	%	Running Time
1×10^3	0.2	57.22
5×10^3	1	57.98
10×10^3	2	59.71
50×10^3	10	71.33
70×10^3	14	81.02
100×10^3	20	97.05

Memory is expressed as number of nodes and as percentage of the size of the original trie.

original one differs only slightly compared to that for the full table. The issue of scalability of the initial computations is, as we have mentioned, an important one because of its domination of the entire runtime. The algorithm is clearly well behaved in this respect.

Table V presents the dependence of the runtime on the amount of extra space available. We can see that the runtime increases more sharply with extra space, compared to the trie size. Again, the extra space is measured in terms of number of nodes.

We also compare our method to extensible hashing. We allow level compression at levels 16 and 24. The reason is that a majority of prefixes are actually 16 or 24 bits long. We were able to compare the two methods only for the Mae-West data set, as the PacBell set was too small for extensible hashing to be applied at the preset levels without inflating the storage requirements out of proportion. The results are comparable only for an extra space of approximately 5×10^4 and 10^5 trie nodes. Extensible hashing is able to achieve an expected performance enhancement of 16.92% (over a level-compressed trie) for 5×10^4 extra nodes, 8% worse than our algorithm. For 10^5 extra nodes, extensible hashing achieves an expected performance enhancement of just 17%, barely half of the corresponding performance enhancement derived from our scheme. However, we would like to note that both the running time and the coding complexity of extensible hashing are extremely low compared to our method.

7. CONCLUDING REMARKS AND ONGOING RESEARCH

In this paper, we have formulated and presented a novel approximation method for the longest matching prefix problem. We have also analyzed the performance of the scheme and have outlined implementation techniques that reduce the runtime and required space. We have demonstrated that our scheme is capable of considerable performance improvements over extensible hashing.

Ongoing research in our group focuses on improvements of the scheme concerning time complexity and approximation ratio. While $O(n \log n)$ is likely to be a lower bound on the time complexity of any algorithm, there is a question as to how much $\xi(d)$ can be reduced. Finally, an incremental version of the algorithm would be of great value. As we have noted in the introduction, the only assumption made concerns the static nature of the routing table. It is desirable that changes in the routing table do not demand a recalculation of the entire structure. Small changes in either the access statistics or the prefix set should cause only minor variations on the existing structure. At the rate routing tables are updated, this may be an important open problem. Updates with respect to changing access patterns would also be of similar importance.

APPENDIX

Proof of Theorem 4.1

We need to prove that during the execution of algorithm A three properties hold. The first is that the density of items outside the knapsack decreases monotonically. The result of this property is that when the knapsack is filled to capacity, there is no item outside the knapsack that has a higher density than an item in the knapsack. The second property is that when the knapsack is filled by A, no item outside the knapsack can replace an item included in the knapsack and produce a valid and better solution. This is not an immediate result of the first property because a lower density item overlapping with a higher density item might have an adverse effect. The final property dictates that there is no combination of lower density items that could replace a combination of higher density items for a valid and better solution.

(1) Let item (v, k) be picked at step i of A. Then, for every item (v', k') that has not been picked yet, we have $\mathit{density}(v, k, i) > \mathit{density}(v', k', i + 1)$.

Proof. If (v, k) and (v', k') do not overlap, it is obvious that the property holds. Otherwise, we distinguish the following cases:

(a) $\mathit{depth}(v') > \mathit{depth}(v)$ and $\mathit{capacity}(v', d, 0) = 0$, where $d = k - \mathit{depth}(v') + \mathit{depth}(v)$.
In this case $\mathit{capacity}(v', k', i) = \mathit{capacity}(v', k', i + 1)$ and $\mathit{benefit}(v', k', i) \geq \mathit{benefit}(v', k', i + 1)$. Therefore, $\mathit{density}(v', k', i + 1) \leq \mathit{density}(v', k', i) < \mathit{density}(v, k, i)$.

(b) $\mathit{depth}(v') > \mathit{depth}(v)$ and $\mathit{capacity}(v', d, 0) > 0$.
In this case, $\mathit{capacity}(v', k', i + 1)$ will be less than $\mathit{capacity}(v', k', i)$. However, if $\mathit{density}(v', k', i + 1) > \mathit{density}(v, k, i)$, there must exist some $(v^*, k^*) \in (v', k')$ such that (v^*, k^*) does not overlap with (v, k)

and $density(v,k,i) < density(v',k',i+1) \leq density(v^*,k^*,i+1) = density(v^*,k^*,i) < density(v,k,i)$, a contradiction.

(c) $depth(v') < depth(v)$ and $capacity(v,d,0) = 0$, where $d = k - depth(v) + depth(v')$.
As in the first case, $capacity(v',k',i+1) = capacity(v,k,i)$, while $benefit(v',k',i+1) \leq benefit(v',k',i)$, therefore, $density(v',k',i+1) \leq density(v',k',i) < density(v,k,i)$.

(d) $depth(v') < depth(v)$ and $capacity(v',d,0) > 0$.
We say that item (u,l) has a *break* at level i, $2 \leq i \leq l$, if there is a node u_1, which has depth $depth(u) + i$, u is one of its ancestors and for which there is no (u_2,l') already in the knapsack with $depth(u_1) > depth(u_2)$ and u_2 is an ancestor of u_1 and $l' + depth(u_2) > depth(u_1)$. The importance of a break is that the benefit of an item is the sum of the benefits of eliminating all of its breaks when level compressing. We will prove by induction that if an item is picked, it has exactly one break or has a zero capacity.

Remark. This is an abstraction of a property that, although never explicitly used, actually characterizes how the items are added to the knapsack. The intuition is that an item expands one level at a time, until its density becomes low enough that another item must be added to the knapsack. The process may return to the same item later, but never does an item expand more than one level, or break, at a time. It is possible to prove the stricter one level at a time property, but the rest of the proofs do not change.

The base case for the induction is the first pick. This will always be a $(u,2)$ and it has one break. Assume that until the ith round only items with at most one break have been picked. The only way the pick at round i can lead to a pick that has more than two breaks in round $i+1$ is by altering the density of an item (u,l) with more than one break so that it becomes more dense than an item (u,l') with one break. We note that the modes of overlapping of cases (a), (b), and (c) do not create such a situation. For the first two cases, it suffices to observe that since (u,k) is placed higher than (u',k'), it removes the tip of the latter and cannot have a favorable effect for an item with two breaks without having the same effect for the less deep item with one break. For the third case, the two items overlap in a full subtrie and the capacity of (u',k') is not altered. It remains to prove that this mode does not allow two-break items to be picked either.

The item picked in the current round is (v,k), and the item overlapping is (v',k'). Suppose, without loss of generality, (v',k') has exactly two breaks (see Figure 6). For items with more breaks, it is easy to prove in the same manner that there is an item with the same root and one less break that has a higher density. Again without loss of generality, we can consider that the last break of any item is at its last level. It can be the case that an item does not have a break there. Then, there is an item of lower depth with at least the same benefit and at most the

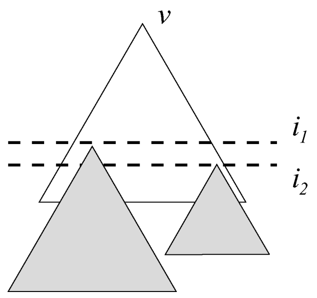


Fig. 6. An item with two breaks. The shaded items are already in the knapsack and represent level compressed subtries. In this case $density(u, i_1, j) > density(u, i_2, j)$, where j is a round where (u, i_1) has not yet been picked.

same capacity, implying its density is larger and any proof that applies to the latter applies to the former. Let $k^* = depth(v) - depth(v')$. We will prove that $density(v', k^*, i+1) > density(v', k', i+1)$ and, therefore, only one-break items are eligible for addition to the knapsack. An item with more breaks must be turned into an item with one break by selecting overlapping items before it can be added to the knapsack.

Let $b_1 = benefit(v', k^*, i)$ and $b_2 = benefit(v, k, i)$. Since (v, k) has only one break and v' is an ancestor of v, $b_1 \geq b_2$. Also, $density(v, k, i) > density(v', k^*, i)$, implies

$$c_1 = capacity(v', k^*, i) > c_2 = capacity(v, k, i).$$

Let $h = k' - k^*$. Then,

$$c_3 = capacity(v', k', i) > 2^h \times c_1,$$

but

$$b_3 = benefit(v', k', i) \leq h \times b_1.$$

Suppose that after picking (v, k), $density(v', k', i+1) > density(v, k, i)$. Let $c_3{}' = capacity(v', k', i+1) = c_3 - c_2 + c$. Then,

$$\frac{benefit(v', k', i+1)}{c_3{}'} \leq \frac{b_3}{c_3{}'} \leq \frac{b_1}{c_1}$$

and

$$density(v', k^*, i) = density(v', k^*, i+1) > density(v', k', i)$$

and only items with one break can be picked. Items with more breaks must contain an item with one break, the density of which is higher. This is simply an extension of the earlier remark in Section 3.2 that $density(v, k, 0) > density(v, k+1, 0)$.

For case (b), when an item with one break is added to the knapsack, all items with the same root must have lower density in the next round. To complete the proof that items are inserted in the knapsack with monotonically decreasing density, we need to show that in this case (case (d)) if (v', k') has only one break at round i, the mode of overlap does not make $density(v', k', i+1) > density(v, k, i)$. As above, we consider only the case where the break is at the last level. Let $b_1 = benefit(v', k', i)$,

$c_1 = \mathit{capacity}(v', k', i)$, $b_2 = \mathit{benefit}(v, k, i)$ and $c_2 = \mathit{capacity}(v, k, i)$. Since $b_1 \geq b_2$ and $\mathit{density}(v, k, i) > \mathit{density}(v', k', i)$, $c_1 > c_2$. After picking (v, k), $\mathit{capacity}(v', k', i+1) = c_1$ and $b_3 = \mathit{benefit}(v', k', i+1) = b_1 - b_2$, which means $\frac{b_3}{c_3} \leq \frac{b_1}{c_1}$ and $\mathit{density}(v', k', i+1) < \mathit{density}(v, k, i)$.

Remark One can rearrange inequalities in the above proofs so that there are no fractions. Then, the fact that densities are decreasing during the execution of the algorithm implies that no capacity can become negative, since this would mean an increase in the density of the corresponding item.

(2) We need to prove that switching an item i in a full knapsack for an item i' outside the knapsack so that the capacity of the knapsack is not exceeded cannot produce a better solution. Observe that the order in which we insert the items in the knapsack does not affect the benefit and the capacity of the overall solution. It only affects the values of the attributes of the items at the round in which they have been added to the knapsack. However, the additive nature of the update procedure dictates that if item j is added in round k, then all of $\mathit{benefit}(j, 0)$ and $\mathit{capacity}(j, 0)$ have been included in the solution, if not by the insertion of j itself, then by the insertion of overlapping with j items during the $k-1$ previous rounds. Therefore, i' can be inserted as the last item after removing i from the solution. Since $\mathit{density}(i) > \mathit{density}(i')$ and $\mathit{capacity}(i) \geq \mathit{capacity}(i')$, $\mathit{benefit}(i) \geq \mathit{benefit}(i')$ and the solution cannot improve by such a switch.

However, there is a technical detail that needs to be addressed—items in the knapsack after removing i may not form a proper solution. Consider an item j in the knapsack, overlapping with i, $\mathit{depth}(\mathit{root}(i)) > \mathit{depth}(\mathit{root}(j))$, and i has been inserted before j. In such a case, the tip of i must be left in the knapsack for a proper solution. The part switched must be the nonoverlapping base of i. If the density of the base is less than that of i', the above argument does not hold. This situation, though, is not possible, due to the monotonicity of the density of items with the same root. Since i was inserted before j, any item i'' with the same root as i but smaller depth and overlapping with j must have been inserted before i. The result is that i consists only of a nonoverlapping base, which is guaranteed to have a larger density than i'.

(3) It remains to prove that no combination of items outside the knapsack can replace a combination of items in the knapsack to create a more profitable solution. Observe that the proof in the previous paragraph implies that a combination of items has density at most as large as that of the item with the highest density and at least that of the item with the lowest density. □

Proof of Theorem 2

Let I be the set of items included in the solution produced by A. Let $i = (v, k)$ be the item with the largest density among those outside I. Then, $\mathit{benefit}(I) \leq \mathit{benefit}(I \cup \{i\})$. Theorem 4.1 implies that $I \cup \{i\}$ is the optimal solution for T with capacity $C + \mathit{capacity}(i)$. We have that $b^* \leq \mathit{benefit}(I \cup \{i\})$. It suffices to prove that $\frac{b}{b+b'} \geq \frac{9}{11}$, where $b' = \mathit{benefit}(i)$.

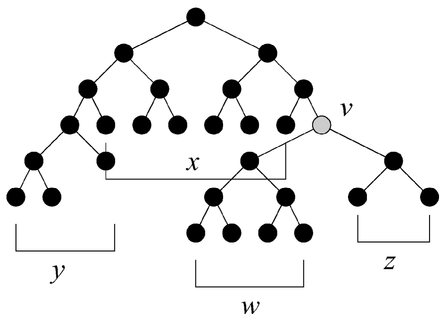


Fig. 7. The trie of Theorem 4.2. The gray node is the root of item i. The letters denote the sum of the weights of the leaves in the corresponding brace.

The trie T can be split into four parts, with regard to i. The first is the subtrie rooted at v and having depth k. This must be at least three levels deep, otherwise $\mathit{capacity}(v, k, j)$ will be at most 2, for every j. Since every node has an even number of children, only even capacities for the knapsack make sense and the residual capacity of the knapsack is at least two nodes. If i has a capacity of two, it would not be outside the knapsack. Furthermore, the larger k is, the larger the contribution of this part of T to b. If (v, k) is the first item that would get in the solution if the capacity was increased, $(v, k-1)$ must already be in, because $\mathit{density}(v, k-1, j) > \mathit{density}(v, k, j)$, for every j. To minimize the ratio $\frac{b}{b+b'}$, k must be 3. Then, for $\mathit{capacity}(v, 3, j) > 1$, where j is the last round of A, $\mathit{capacity}(v, 3, 0) \geq \mathit{capacity}(v, 2, 0) + 2$.

The second part of T is the one consisting of all the subtries rooted at a descendant of v. The effect of this part of T on the solution returned by A is the combination of the effect of all the items rooted at a node that is a descendant of v. This set of items can be further split into those overlapping with (v, k) and those that do not. If there are any overlapping items in the solution, the effect of b' on the ratio is reduced. To minimize the ratio, there must be no such items in the solution.

The third part of T is the one above the level of v. To allow for $(v, 3)$ to be i, there must be no overlap between $(v, 3)$ and items in the knapsack rooted at some node of depth less than $\mathit{depth}(v)$. To have no such overlap, v must be at depth at least 3. If it is at depth 0, it will be the root and it will be the first item to be picked. If it is at depth 1 or 2, $(\mathit{root}, 3)$ will be picked and since it overlaps with $(v, 3)$, the latter cannot be i. The reason is that if $(\mathit{root}, 2)$ is picked, but not $(\mathit{root}, 3)$, the capacity of the knapsack cannot be more than 2. For A to produce a suboptimal solution, the fourth part of the trie, described in the next paragraph, must exist. In this case, there are four nodes, at most, missing from $(\mathit{root}, 3)$. If the capacity of the knapsack is four nodes or more, the density of $(\mathit{root}, 3)$ must be the highest among the initial items and it should be the first item in the knapsack. On the other hand, the larger this part of the trie, the weaker the contribution of b' to the ratio.

The fourth part of the T consists of all the items whose root is not an ancestor of v and do not overlap with i. To minimize the ratio, the contribution of these items to the solution must be minimized. However, it cannot be 0 because this would imply the solution, as it is, is optimal. Therefore, there should be only one item from this set in the knapsack, it should have nonzero capacity and its depth should be 2. This implies that its capacity is 2.

All of the above cases are summarized in Figure 7. Consider that the capacity of the knapsack is 4. If the capacity is less than 4, the solution produced by the greedy algorithm is optimal and if it is 12 or more the greedy algorithm will produce an optimal solution, as well. For any capacity between 6 and 11, i will end up in the knapsack. For capacity 4, $b = 2 \times (w + x + y + z) + w + y + z$ and $b' = w$. To minimize the ratio, w must be as large as possible. Since the item with benefit y and capacity 2 has entered the knapsack before i, which has benefit w and capacity 4, $w \leq 2 \times y$, therefore, $w = 2 \times y$ to minimize the ratio. Finally, the upper bound for $\frac{b}{b+b'}$ is $\frac{9\times z+q}{11\times z+q}$, where q is independent of z. This ratio is at least $\frac{9}{11}$.

ACKNOWLEDGMENTS

We would like to thank Prof. M. Waldvogel for his help in providing us with input data for our experiments.

REFERENCES

ANDERSSON, A. AND NILSSON, S. 1994. Faster searching in tries and quadtrees: An analysis of level compression. In *Proceedings of ESA '94*.

BREMLER-BARR, A., AFEK, Y., AND HAR-PELED, S. 1999. Routing with a clue. In *Proceedings of SIGCOMM '99*.

CHANDRANMENON, G. V. G. AND VARGHESE, G. 1996. Trading packet headers for packet processing. *IEEE/ACM Transactions on Networking 4*, 2 (Apr.), 141–152.

CHEUNG, G. AND MCCANNE, S. 1999. Optimal routing table design for IP address lookups under memory constraints. In *Proceedings of INFOCOM '99*.

CHEUNG, G., MCCANNE, S., AND PAPADIMITRIOU, C. 1999. Software synthesis of variable-length code decoder using a mixture of programmed logic and table lookups. In *Proceedings of DCC '99*.

CRESCENZI, P., DARDINI, L., AND GROSSI, R. 1999. IP address lookup made fast and simple. In *Proceedings of ESA '99*.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM '97*.

GUPTA, P., LIN, S., AND MCKEOWN, N. 1998. Routing lookups in hardware at memory access speeds. In *Proceedings of INFOCOM '98*.

LABOVITZ, C., MALAN, G., AND JAHANIAN, F. 1997. Internet routing instability. In *Proceedings of SIGCOMM '97*.

LAMPSON, B., SRINIVASAN, V., AND VARGHESE, G. 1998. IP lookups using multiway and multicolumn search. In *Proceedings of INFOCOM '98*.

MARTELLO, S. AND TOTH, P. 1990. *Knapsack Problems*. Wiley, New York.

MCAULEY, A., TSUCHIYA, P., AND WILSON, D. n.d. Fast multilevel hierarchical routing table using content-addressable memory. U.S. Patent serial number 034444.

NARLIKAR, G. AND ZANE, F. 1991. Performance modeling for fast IP lookups. In *Proceedings of SIGMETRICS '91*.

NEWMAN, P., MINSHALL, G., AND HUSTON, L. 1997. IP switching and gigabit routers. *IEEE Communications Magazine 35*, 1 (Jan.), 64–69.

NILSSON, S. AND KARLSSON, G. 1998. Fast address lookup for internet routers. In *IFIP International Conference of Broadband Communications Conference Proceedings*.

PARULKAR, G., SCHMIDT, D., AND TURNER, J. 1995. IP/ATM: A strategy for integrating IP with ATM. In *Proceedings of SIGCOMM '95*.

REKHTER, Y., DAVIE, B., KATZ, D., ROSEN, E., SWALLOW, G., AND FARINACCI, D. n.d. Tag switching architecture overview. Internet Draft.

SIKKA, S. AND VARGHESE, G. 2000. Memory-efficient state lookups with fast updates. In *Proceedings of SIGCOMM '00*.

SKLOWER, K. 1991. A tree-based routing table for Berkeley Unix. In *Proceedings of the 1991 Winter Usenix Conference*.

SRINIVASAN, V. AND VARGHESE, G. 1998. Faster IP lookups using controlled prefix expansion. In *Proceedings of SIGMETRICS '98/Performance '98*.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high-speed IP routing lookups. In *Proceedings of SIGCOMM '97*.

Received September 2004; revised March 2005; accepted March 2005