Adaptive Data Structures for IP Lookups

Ioannis Ioannidis, Ananth Grama, and Mikhail Atallah
Department of Computer Sciences,
Purdue University,
W. Lafayette, IN 47907.
{ioannis, ayg, mja}@cs.purdue.edu

Abstract— **The problem of efficient data structures for IP lookups has been well studied in literature. Techniques such as LC tries and Extensible Hashing are commonly used. In this paper, we address the problem of generalizing LC tries and Extensible Hashing, based on traces of past lookups, to provide performance guarantees for memory sub-optimal structures. As a specific example, if a memory-optimal (LC) trie takes 6MB and the total memory at the router is 8MB, 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 $\frac{9\times b}{22}$ 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 $O(\xi(d)n \times \log n)$ 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 demonstrate experimentally the performance and scalability of the algorithm on actual routing data. We also show that our algorithm significantly outperforms Extensible Hashing for the same amount of memory.**

I. Introduction and Motivation

The problem of developing efficient data structures for IP lookups is an important and well studied one. Given an address, the 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, find the 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 requirement, called *level-compression*, is described in [14]. 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 $\log^* n$, where n is the size of the original trie, under reasonable assumptions for the probability distribution of the input. In [12], a generalization of level-compression, usually referred as *extensible hashing*, was presented. In extensible hashing, certain levels of the trie are filled and subsequently level-compressed. These levels are selected to be 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 of access. We are also given a constraint on the total router memory, say, 8MB. If the trie for the prefixes requires only 6MB of memory, we would like to build hash tables in the trie to best utilize the 2MB 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 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.

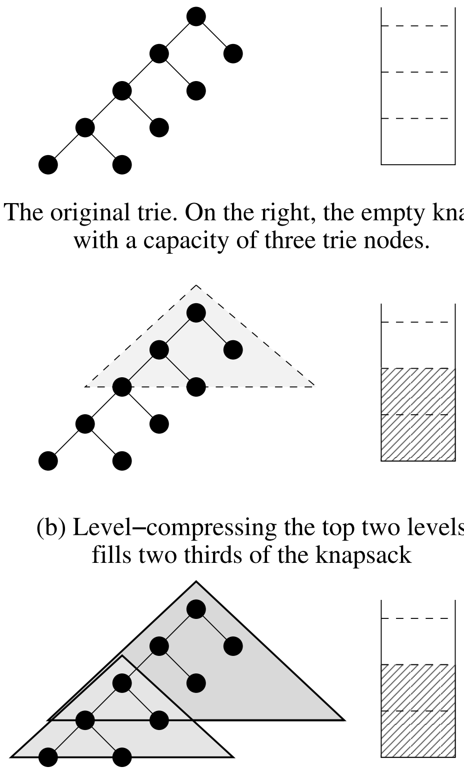
We can formulate a generalization of the level-compression and extensible hashing schemes as a variation of the knapsack problem. The items to be included in the knapsack are subtries. The gain of an item is the reduction in average lookup time that results from level-compressing this subtrie and its cost is a function of the number of missing leaves in the subtrie (i.e., 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 precisely defined 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

0-7803-7753-2/03/$17.00 (C) 2003 IEEE

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 [19].

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.

II. Overview of the Algorithm and Related Research


(a) The original trie. On the right, the empty knapsack, with a capacity of three trie nodes.

(b) Level–compressing the top two levels fills two thirds of the knapsack

(c) The remaining items are too large for the knapsack

Fig. 1. A simple example of the greedy algorithm

We 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 between 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 (Figure 1). Although greedy algorithms are generally 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. Selecting a subtrie for level-compression has a cascading effect on items on the path to the root and the leaves. This complicates the algorithm, proof of bound, and implementation very significantly. We show that the greedy algorithm can approximate the optimal solution within a factor of $\frac{9}{22}$. The approximation is with respect to the original, non-level-compressed 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 $\frac{9\times b}{22}$ fewer memory accesses on average. We note that the problem is known to be NP-complete ([18]).

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 the scheme's usefulness. We describe an implementation with a time complexity of $O(\xi(d)n \times \log n)$ 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 give 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 are presented as well.

A. 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 [1]. True to the spirit of UNIX, simplicity is not sacrificed for performance. In [2], [3], [4], 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 [10], where lookups are accelerated using memory placement and pipelining. Some protocol-based solutions have emerged ([5], [6], [4], [7], [8], [9]), 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 ([13], [12], [11], [15], [16]). The advantage of these is their transparency to protocol and to advances in hardware platforms. In [14], the original level-compression scheme was described. The only effort we are aware of on formulating a generalization of level-compression to include memory constraints and providing a solution was presented by Cheung and McCanne [17]. They formalize memory constraints in the form of an arbitrary memory hierarchy. Cheung et al. [18], also show that the problem is NP-complete even for one-level memory and present a simple, dynamic-programming, pseudo-polynomial time algorithm. An approximation using Lagrange multipliers is also described, although no constant bound on the error is derived for this approximation scheme.

III. 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 problem is the dynamic nature of the items. In knapsack, selecting an 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 run-time 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 simple greedy algorithm, which works along these lines and derive an efficient implementation.

In the following we assume that for each leaf of the trie we have access statistics available to us. This information corresponds to weights for the different access paths. 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.

A. *Definitions*

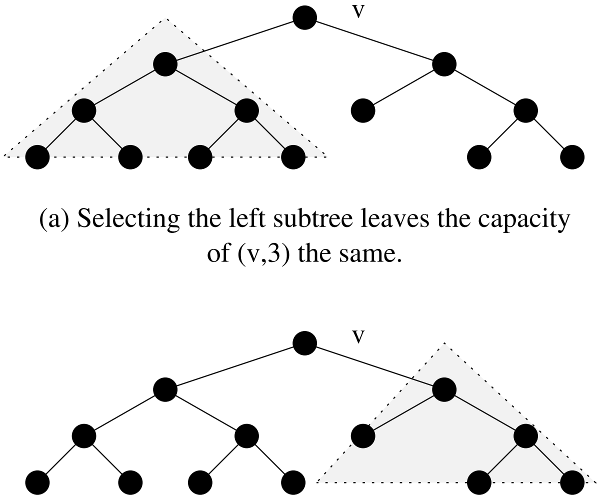
We want the trie structure to reflect the expected number of accesses going through a node. We formalize the notion of a trie with access statistics for the addresses. An internal node must be accessed at least as many times as its two children combined.

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.

Informally, adding an item i to the knapsack corresponds to a decision to level-compress the subtrie corresponding to i. We can identify a subtrie by its 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 2.



(a) Selecting the left subtree leaves the capacity of (v,3) the same.

(b) Selecting the right subtrie makes the capacity of (v,3) zero.

Fig. 2. Item $(v, 3)$ has different capacities in (a) and (b)

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.

Definition 3.3: For every $i = (v, k)$, we define $capacity(v, k, 0)$ as 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.

Definition 3.4: For every $i = (v, k)$, we define $benefit(v, k, 0) = \sum_{l=2}^{k} \sum_{u \in C(v,l)} w(u)$, 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 cannot be defined. Also, a rule must be provided to resolve the ties of items with equal densities. For these reasons, we formally define $density$ as any mapping from the set of items to a set of cardinality $(n \times d)^2$. 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) $benefit(v_1, k_1, i_1) \times capacity(v_2, k_2, i_2) <$ $benefit(v_2, k_2, i_2) \times capacity(v_1, k_1, i_1)$
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 $capacity(v_1, k_1, i_1) > capacity(v_2, k_2, i_2)$.

3) $capacity(v_1, k_1, i_1) = capacity(v_2, k_2, i_2)$, $benefit(v_1, k_1, i_1) \times capacity(v_2, k_2, i_2) = benefit(v_2, k_2, i_2) \times capacity(v_1, k_1, i_1)$ and $depth(v_1) > depth(v_2)$.
4) $capacity(v_1, k_1, i_1) = capacity(v_2, k_2, i_2)$, $benefit(v_1, k_1, i_1) = benefit(v_2, k_2, i_2)$, $depth(v_1) = depth(v_2)$ and $v_1 < v_2$.

We assume there is a linear ordering of the nodes (for example, a depth-first ordering, although any ordering suffices). We note that densities are ordered according to the ratio whenever possible. If there is a tie, the item with the lowest capacity takes precedence. If there is still a tie, an item is arbitrarily selected. Note that we have not provided an ordering for densities of the same item during different rounds because such a comparison never takes place.

In the rest of the paper, for the sake of simplicity, we will 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.

B. The Update Rules

We describe the rules by which $capacity(v, k, i)$ and $benefit(v, k, i)$ are obtained from $capacity(v, k, i-1)$ and $benefit(v, k, i-1)$ respectively, for all $i > 0$. We assume that the i-th item added to the knapsack is (v', k'). We first define formally the situation in which two items affect each other.

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 $depth(v_1) + k_1 - 1 \geq depth(v_2)$ or v_1 is a descendant of v_2 and $depth(v_2) + k_2 - 1 \geq depth(v_1)$. Furthermore, if $depth(v_1) + k_1 \geq depth(v_2) + k_2$, we say that (v_1, k_1) contains (v_2, k_2) and we write $(v_2, k_2) \in (v_1, k_1)$.

The *overlaps* relation is symmetric. 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, $benefit(v, k, i) = benefit(v, k, i-1)$ and $capacity(v, k, i) = capacity(v, k, i-1)$.

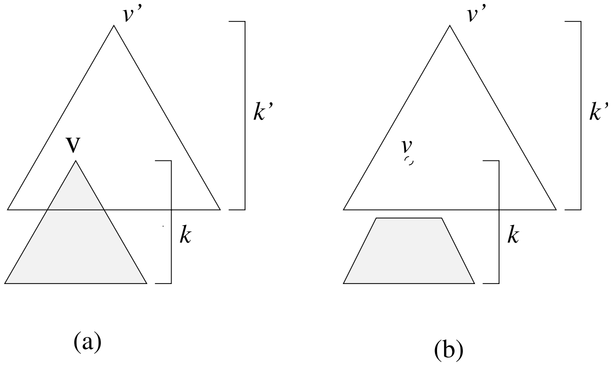


Fig. 3. Updating overlaps: (a) Before updating (v, k); (b) After updating.

- $depth(v) > depth(v')$ (Figure 3)
  If $(v, k) \in (v', k')$, $capacity(v, k, i) = benefit(v, k, i) = 0$. This is because level-compressing a subtrie level-compresses all the subtries included in it. Otherwise, let $l = depth(v') + k' - depth(v)$. 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, $benefit(v, k, i) = \sum_{u \in C(v,l)} benefit(u, k-l, i)$. 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, $capacity(v, k, i) = \sum_{u \in C(v,l)} capacity(u, k-l, i)$.
- $depth(v) = depth(v')$
  If $k < k'$, $(v, k) \in (v', k')$ and $capacity(v, k, i) = benefit(v, k, i) = 0$. Otherwise, $(v', k') \in (v, k)$ and $capacity(v, k, i) = capacity(v, k, i-1) - capacity(v', k', i-1)$ and $benefit(v, k, i) = benefit(v, k, i-1) - benefit(v', k', i-1)$.

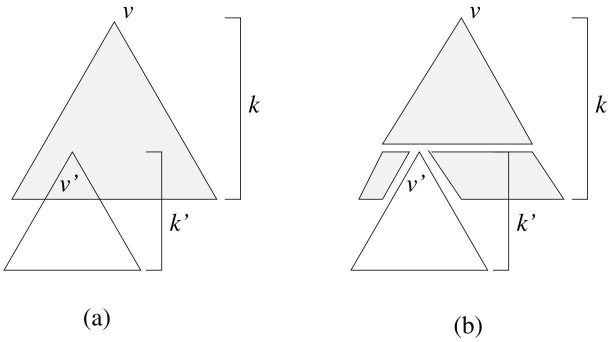


Fig. 4. Updating overlaps: (a) Before updating (v, k); (b). After updating.

- $depth(v) < depth(v')$ (Figure 4)
  This case is the dual of the first one. Let $l = depth(v) + k - depth(v')$. We have:

$$benefit(v, k, i) = benefit(v, l, i-1) + \sum_{u \in C(v,l) - \{v'\}} benefit(u, k-l, i-1)$$

and

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

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.

IV. 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

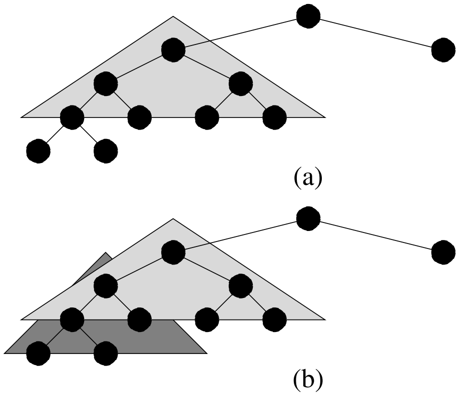


Fig. 5. Incorrect view of the item set. In (a), the shaded item is selected. In (b), once the shaded item at the top is selected, algorithm A cannot see the bottom left item.

added exceeds the available knapsack capacity. Algorithm A would produce the optimal solution whenever the knapsack is filled to capacity, if it were not for a flaw in the way it reflects the level-compression process. An illustration of this flaw is given in Figure 5. This problem is a result of how the selection of an item binds all the contained items to either expand all together or not expand at all. A subsequent overlap may allow such items to expand independently but A has already committed to the mistaken view of the item set. For the following theorem we will assume that such errors do not affect the solution produced by A. We formalize this assumption in the following.

Assumption 1: If item i is selected in round j, there is no item i_1 such that $i_1 \in i$ and there exists item i_2 not overlapping with i that is not selected in the knapsack when A terminates after k rounds and such that $capacity(i_2, k) \leq capacity(i_1, j)$ and $benefit(i_2, k) > benefit(i_1, j)$.

Theorem 1: If algorithm A fills knapsack K to capacity and Assumption 1 holds, the solution induced is optimal.

Remark: We prove subsequently that if the above assumption does not hold, the solution produced by A has at worst half the benefit of the optimal one.

Proof: We will prove that during the execution of A, three properties hold.

1) Let item (v, k) be picked at step i of A. Then, for every item (v', k') which has not been picked yet, we have $density(v, k, i) > density(v', k', i+1)$. In other words, the densities of the items inserted in the knapsack are monotonically decreasing.
   If (v, k) and (v', k') do not overlap, it is obvious that the property holds. Otherwise, we distinguish the following cases:
   a) $depth(v') > depth(v)$ and $capacity(v', d, 0) = 0$, where $d = k - depth(v') + depth(v) - 1$.
      In this case $capacity(v', k', i) = capacity(v', k', i + 1)$ and $benefit(v', k', i) \geq benefit(v', k', i+1)$. Therefore, $density(v', k', i+1) \leq density(v', k', i) < density(v, k, i)$.
   b) $depth(v') > depth(v)$ and $capacity(v', d, 0) > 0$.
      In this case, $capacity(v', k', i+1)$ can be less than $capacity(v', k', i)$. However, if $density(v', k', i+1) > density(v, k, i)$, 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 $d = k - depth(v) + depth(v') - 1$.
      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 $capacity(v', d, 0) > 0$.

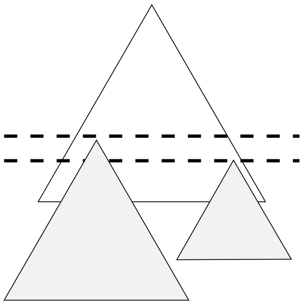


Fig. 6. An item with two breaks. The shaded items are already in the knapsack and represent level-compressed subtries

We say that item (u, l) has a *break* at level i, $i \leq l$, if there is a node u_1, which has depth $depth(u) + i$ and 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. If it has no breaks, it has no benefit and there is no way it can be picked if it has a non-zero capacity.
The base case is the first pick. This will always be an $(u, 1)$ and it has one break. Assume that until the i-th 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 observe that the modes of overlapping of the three previous cases do not create such a situation. It remains to prove that this mode doesn't either.
The item picked in the current round is (v, k) and the item overlapping is (v', k'). Suppose (v', k') has at least two breaks. Without loss of generality,

we can consider the last break of any item is at its last level. It can be the case that an item doesn't have a break there. Then, there is a less deep item with at least the same density and at most the 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 $density(v', k', i+1) > density(v, k, i)$ implies that $density(v', k^*, i) > density(v, k, i)$, a contradiction.

Let $b_1 = benefit(v', k^*, i)$ and $b_2 = benefit(v, k, i)$. Since (v, k) has only one break, $b_1 \geq b_2$. 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 $density(v', k^*, i) = density(v', k^*, i+1) > density(v, k, i)$.

Since $density(v', l, i+1) > density(v, k^*, i+1)$ and $density(v', l, i+1) = density(v', l, i)$, $l < k^*$, only items with one break will be picked. It remains to be shown that if (v', k') has only one break at round i, the mode of overlap of this case 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 $b_1 = benefit(v', k', i)$, $c_1 = capacity(v', k', i)$, $b_2 = benefit(v, k, i)$ and $c_2 = capacity(v, k, i)$. Since $b_1 \geq b_2$ and $density(v, k, i) > density(v', k', i)$, $c_1 > c_2$. After picking (v, k), $c_3 = capacity(v', k', i+1) = c_1 - c_2$ and $b_3 = benefit(v', k', i+1) = b_1 - b_2$, which means $\frac{b_3}{c_3} \leq \frac{b_1}{c_1}$ and $density(v', k', i+1) < density(v, k, i)$.

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 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 $benefit(j, 0)$ and $capacity(j, 0)$ 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 $density(i) > density(i')$ and $capacity(i) \geq capacity(i')$, $benefit(i) \geq benefit(i')$ and the solution cannot improve by such a switch.

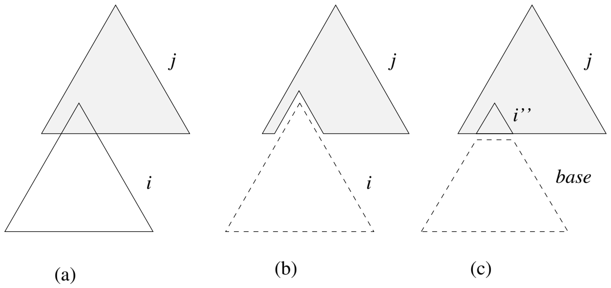


Fig. 7. (a) Items i and j; (b) Removing i creates an illegal item; (c) Removing the base of i.

However, there is a problem – items in the knapsack after removing i may not form a proper solution. Consider an item j in the knapsack, overlapping with i, $depth(root(i)) > depth(root(j))$, 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 non-overlapping base of i. If the density of the base is less than that of i', the above argument does not apply. This situation, though, is not possible, due to the monotonicity of the density of items with the same root. Because, 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 non-overlapping base which is guaranteed to have a larger density than i'. The above cases are illustrated in Figure 7.

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. This follows from the observation 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.

A. Approximation ratio

In this section we consider only tries whose nodes have either two or no children. This is the case for tries representing routing tables, since any linear subtrie can be compressed into a single node. This technique of path compression is routinely used to decrease the size of a trie. We will first prove that for such tries, if the solution produced is not affected by the flaw in the representation of the level-compression process by A, the benefit of this solution cannot be less than $\frac{9}{11}$ of that of the optimal solution.

Theorem 2: Let b be the benefit of the solution produced by A for a trie T corresponding to a routing table with capacity C and b^* the benefit of the optimal solution for T with capacity C. Then, $\frac{b}{b^*} > \frac{9}{11}$.

Proof: 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, $benefit(I) \leq$

$benefit(I\cup\{i\})$. Theorem 1 implies that $I\cup\{i\}$ is the optimal solution for T with capacity $C + capacity(i)$. We have that $b^* < benefit(I\cup\{i\})$. It suffices to prove that $\frac{b}{b+b'} > \frac{9}{11}$, where $b' = benefit(i)$.

The trie T can be split in 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 $capacity(v, k, j)$ will be at most 2, for every j. 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 $density(v, k-1, j) > density(v, k, j)$, for every j. To minimize the ratio $\frac{b}{b+b'}$, k must be 3. Then, for $capacity(v, 3, j) > 1$, where j is the last round of A, $capacity(v, 3, 0) \geq capacity(v, 2, 0) + 2$.

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 to the solution returned by A is the combination of the effect of all the items rooted at a node which is a descendant of v. This set of items can be further split in 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 $depth(v)$. 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, $(root, 3)$ will be picked and since it overlaps with $(v, 3)$, the latter cannot be i. On the other hand, the larger this part of the trie, the weaker the effect 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 non-zero capacity and its depth should be two. This implies its capacity is two. Also, since it has density larger than $(v, 3)$, its benefit must be at least half of that of $(v, 3)$.

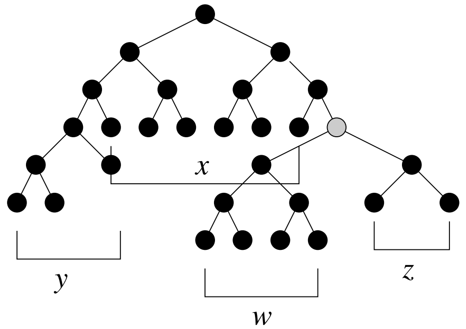


Fig. 8. The trie of theorem 2. The grey node is the root of item i. The letters denote the sum of the weights of the leaves in the corresponding brace.

All the above are summarized in Figure 8. Consider that the capacity of the knapsack is four. If the capacity is less than four, the solution produced by the greedy algorithm is optimal and if it is twelve or more the greedy algorithm will produce an optimal solution, as well. For any capacity between six and eleven, i will end up in the knapsack. For capacity four, $b = 2 \times (w + x + y + z) + x + y + z$ and $b' = b + x$. To minimize the ratio, x must be as large as possible. We know that $x \leq 2 \times z$, therefore, $x = 2 \times z$. 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 $\frac{9}{11}$.

We can prove that if A is indeed affected by the flaw for a certain instance of the problem, the benefit of the produced solution is at least $\frac{9}{22}$ of that of the optimal solution. It suffices to prove that whenever A fills the knapsack to capacity, the induced solution has at least half the benefit of the optimal one.

Theorem 3: If algorithm A fills knapsack K to capacity, the benefit of the solution induced is at least half of that of the optimal solution.

Proof: Consider an item i to which A has committed incorrectly and has been included in the knapsack at step j. Without loss of generality, let i consist of two simple items i_1 and i_2, such that $density(i_1, j) > density(i_2, j)$. Let i' be an item such that $density(i, j) > density(i', j)$ but $density(i_2, j) < density(i', j)$. Also, let $capacity(i', j) \leq capacity(i_2, j)$ and $benefit(i', j) > benefit(i_2, j)$. Obviously, A will produce a suboptimal solution, because adding i_1 and i' to the knapsack would produce a larger combined benefit, without exceeding the capacity of i. It follows that $benefit(i_1, j) + benefit(i', j) \leq 2 \times benefit(i, j)$ and there is no solution having benefit more than two times that of the solution returned by A.

Corollary 1: The approximation ratio of A is $\frac{9}{22}$.

V. Complexity

A naive implementation of the above algorithm would yield a runtime of $O(n^2 \times d^2)$ and would use $O(n \times d)$ 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 $O(n \times d)$ time. Each update operation would take $O(n)$ time and it can be proved there are $O(d)$ such amortized operations in each step. There are $O(n \times d)$ steps, hence the running time. A more efficient implementation uses only linear space and has a running time of $O(n \times d^2 \log n)$.

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 takes time $O(\log n)$, as mentioned. Updating the overlapping items might take $O(n^2 \times d^2)$ 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 at the path from m to the root of the trie fall in this category. We split the update operations in two categories. The first is the one involving the items mentioned above. The second involves items having m as an ancestor of their root. The first category produces $O(d)$ updates at each step. There are $O(n \times d)$ steps, for a total of $O(n \times d^2)$ 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, $O(n \times d^2)$ updates of the second category can be executed. In all, there are $O(n \times d^2)$ 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 $S = C(v, k) \cap C(v', k - depth(v') + 1)$. 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 where $depth(v) \geq depth(v')$ 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 neighbor 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 trees on each level of the trie. Both have the property that their leaves are the nodes of the trie belonging to the corresponding level, ordered from left to right. 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. The information kept in the trie nodes and the auxiliary trees built on each level of the trie do not change over the execution of the algorithm. Therefore, the total space needed for the algorithm is $O(n)$ and the total time asymptotically remains the same. We note that each tree has depth $O(\log n)$.

We can use the auxiliary structures to do an update in $O(\log n)$ time. Let w and x be the leftmost and rightmost nodes of S. We can get the information we need for all the nodes in S by traversing the auxiliary trees. For the weights we can first traverse the path from the root to w and then the path from the root to x. Spending $O(1)$ time on each node on these paths we can derive the information in $O(\log n)$ time. A similar process can derive the information for the number of missing nodes from the second tree.

In this manner, the total time is $O(n \times d^2 \times \log n)$ and the total space needed is $O(n)$.

VI. 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 use a Mae-West routing table. The size of the resulting trie was a little over 5×10^5 nodes. Besides the improvement of the expected lookup time, we measure the running time relative to the routing table size and relative to the parameter of extra space. Finally, we apply extensible hashing on levels 16 and 24 of the trie and compare its performance to that of our algorithm. All experiments were run on 2GHz P4 with 512MB RAM.

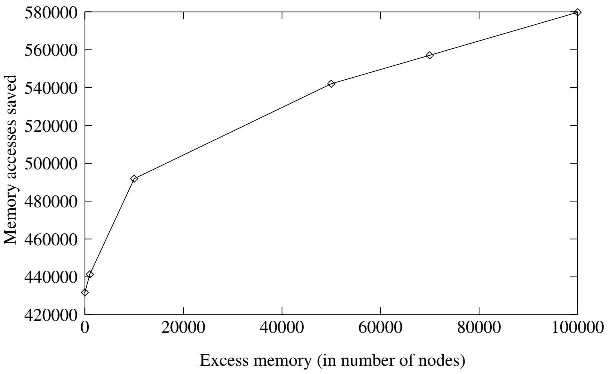


Fig. 9. Performance of the algorithm in terms of memory accesses saved over a level compressed trie.

Table I contains values of the performance enhancement as a percentage of that of simple level-compression for varying amounts of excess memory. In Figure 9 these values are illustrated graphically. 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 II presents the runtime of the algorithm for different sizes of the routing table, with no extra space allowed. The depth of the resulting trie has been kept constant in all cases. This measurement is necessary since a large part of the runtime

Extra Space	Performance Enhancement
1×10^3	2.21%
5×10^3	11.8%
10×10^3	13.9%
50×10^3	25.52%
70×10^3	29.01%
100×10^3	34.27%

TABLE I

EXPERIMENTAL MEASUREMENTS OF IMPROVEMENT IN NUMBER OF LOOKUPS (AS A PERCENTAGE OF THOSE OF A LEVEL COMPRESSED TRIE) WITH INCREASING AMOUNTS OF EXTRA MEMORY.

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 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.

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

TABLE II

EXPERIMENTAL MEASUREMENTS (IN SECS) OF THE RUNTIME FOR ROUTING TABLES OF DIFFERENT SIZES. THE FIRST COLUMN INDICATES THE SIZE OF THE TABLE AS A PERCENTAGE OF THE SIZE OF THE FULL ROUTING TABLE.

Table III 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.

Extra Space	Running Time
1×10^3	57.22
5×10^3	57.98
10×10^3	59.71
50×10^3	71.33
70×10^3	81.02
100×10^3	97.05

TABLE III

EXPERIMENTAL MEASUREMENTS (IN SECS) OF THE RUNTIME FOR DIFFERENT AMOUNTS OF EXTRA SPACE ALLOWED. THE FIRST COLUMN INDICATES THE NUMBER OF NODES FILLED.

Finally, we 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. The results were comparable only for an extra space of approximately 5×10^4 and 10^5 trie nodes. Extensible hashing was 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 achieved an expected performance enhancement of just 17%, barely half of the corresponding performance enhancement derived from our scheme.

VII. 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 the most important open problem in this area. Updates with respect to changing access patterns would also be of similar importance.

Acknowledgements

This work is supported in part by National Science Foundation grants EIA-9806741, ACI-9875899, ACI-9872101, EIA-9903545 and ISS-0219560, contract N00014-02-1-0364 from the Office of Naval Research, and by Computing Research Institute at Purdue University, and sponsors of the Center for Education and Research in Information Assurance and Security. We would also like to thank Prof. M. Waldvogel for his help in providing us with input data for our experiments.

References

[1] K. Sklower, ”A Tree-Based Routing Table for Berkeley Unix”, Proceedings of the 1991 Winter Usenix Conference.

[2] P.Gupta, S.Lin, N.McKeown, ”Routing Lookups in Hardware at Memory Access Speeds” Infocom 98, vol.3, pp.1240-7, 1998.

[3] A. McAuley, P. Tsuchiya, D. Wilson, ”Fast Multilevel Hierarchical Routing Table Using Content-Addressable Memory”, U.S. Patent Serial Number 034444.

[4] P. Newman, G. Minshall, L. Huston, ”IP Switching and Gigabit Routers”, IEEE Communications Magazine (January)

[5] G. Chandranmenon, G. Varghese, ”Trading Packet Headers for Packet Processing”, IEEE/ACM Transactions on Networking (April).

[6] C. Labovitz, G. Malan, F. Jahanian, ”Internet Routing Instability”, Proceedings of SIGCOMM ’97 (October)

[7] G. Parulkar, D. Schmidt, J. Turner, ”IP/ATM: A Strategy for Integrating IP with ATM”, Proceedings of SIGCOMM ’95 (October).

[8] Y. Rekhter, B. Davie, D. Katz, E. Rosen, G. Swallow, D. Farinacci, ”Tag Switching Architecture Overview”, Internet Draft.

[9] A. Bremler-Barr, Y. Afek, S. Har-Peled, ”Routing with a Clue”,

[10] S. Sikka, G. Varghese, ”Memory-Efficient State Lookups with Fast Updates”, Proceedings of SIGCOMM 2000, pp. 335-347.

[11] P. Crescenzi, L. Dardini, R. Grossi, ”IP Address Lookup Made Fast and Simple”, Dipartmento Di Informatica, Università Di Pisa, Technical Report TR-99-01.

[12] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, ”Scalable High-Speed IP Routing Lookups”, Proceedings of SIGCOMM ’97 (October).

[13] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, ”Small Forwarding Tables for Fast Routing Lookups”, Proceedings of SIGCOMM ’97 (October)

[14] S. Nilsson, G. Karlsson, ”Fast Address Look-Up for Internet Routers”, Proceedings of IEEE Communications Magazine (January).

[15] V.Srinivasan, G.Varghese, ”Faster IP Lookups using Controlled Prefix Expansion”, Proceedings of the ACM SIGMETRICS ’98/Performance ’98.

[16] B.Lampson, V.Srinivasan, G.Varghese, ”IP Lookups using Multiway and Multicolumn Search”, Infocom 98, vol.3, pp.1248-56, 1998.

[17] G. Cheung, S. McCanne, ”Optimal Routing Table Design for IP Address Lookups Under Memory Constraints”, Proceedings of INFOCOM ’99, pp. 1437-1444

[18] G.Cheung, S.McCanne, C.Papadimitriou, ”Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups”, DCC ’99, pp. 121-130.

[19] S. Martello, P. Toth, ”Knapsack Problems”, J. Wiley & Sons, Chichester, 1990