H-Tree: A data structure for fast path-retrieval in rooted trees.

Akram Shehadi Saynez
saganus00@gmail.com

María Josefa Somodevilla
mariasg@solarium.cs.buap.mx

Manuel Martín Ortiz
mmartin@solarium.cs.buap.mx

Ivo H. Pineda
ipineda@solarium.cs.buap.mx

Facultad de Ciencias de la Computación, Benemérita Universidad Autónoma de Puebla
Puebla, México

Abstract

When repeatedly searching on a Depth-First or Breath-First Search fashion, some nodes are visited many times, as every search requires to start at the root. By providing a way to arrange the data in a different manner and using a hash table as the primary data structure, it is possible to get average lookup times of a path, from the root to any node in constant time, although a disadvantage arises when looking at the memory needs of the approach. We propose a structure called H-Tree created to reduce the time to extract a path from a rooted tree.

Keywords: *Rooted tree, Huffman coding, path retrieval, hash table, depth-first search.*

1. Introduction

Some algorithms which use trees as their principal data structures often need new ways to traverse them, focusing on the optimization of a particular goal. Sometimes the applications that use rooted trees, need too, the ability to extract the path from the root to any given node. Traditional traversing methods (namely Depth-First and Breath-First searches), when trying to find a route to a node starting at the root, are roughly comparable to finding a house in a city, by standing in the center ("root") of it and walking every possible street until the location is found. For some problems this is the best solution, only admitting some degree of optimization, but we believe that for some others there is a better approach.

The basic idea behind this algorithm is equivalent to first building a map of the city, when the searching for several houses is needed; by walking the whole city only once, assigning an "address" to each lot, and using it as a guide to find landmarks, then the location of a particular site can be done much faster. The guide will tell exactly which "street" the required lot is located in, and its distance from the "root" city (upon which the map was built), effectively cutting the need to "walk" each and every possible street, every time.

We propose a method to build a "cache" of the paths from the root of a tree to every other node, so as to avoid visiting certain nodes (namely the shallow ones, specially the root) repeated times.

This paper is organized as follows. Section 2 explains the motivation behind this work. In Section 3 we detail the algorithm created, and describe the analysis made on it in Section 4. Section 5 shows results from different experiments performed, while Section 6 presents possible improvements to the method. Finally, the references are on Section 7.

2. Preliminaries

This section explains the motivation behind the technique developed.

We propose a mechanism for transforming any rooted tree into a data structure we call an H-Tree, which is basically a hash table representing a tree, with each entry in the table being a node in the tree associated with it's path from the root.

By generating an H-Tree from a given rooted tree, the need to perform DFS- or BFS-like traversing methods could be avoided in some cases; by providing more memory, the path finding on a tree is reduced to a lookup operation over a hash table structure.

Huffman coding, used in a lot of compression algorithms, can benefit from such approach. By building an H-Tree out of the Huffman tree obtained from some data set, it is possible to locate the corresponding symbol to any given codeword, at a faster speed, therefore, having all the possible symbols and their Huffman codes stored in a hash-table, allows the

0-7695-2899-6/07 $25.00 © 2007 IEEE
DOI 10.1109/ENC.2007.26

algorithm to expand any bit stream in $\Omega(n)$ look-up operations, where n is the total number of characters in it.

A key feature of this method is that the generated H-Tree can be used multiple times, without the need to build it every time it is used. For example, the H-Tree for the Huffman codes corresponding to the English alphabet frequencies, can be created once and stored in a file to be loaded by the decompressing application, such that only the bit stream needs to be transmitted or stored.

In [2] the data structure presented is space-optimal for the problem of 2-dimensional range searching and answer queries in a time that is within a very small additive term of the optimum, whereas in [3] a novel trie-based distributed data structure called a Prefix Hash Tree for the solution of 1-dimensional range queries over a Distributed Hash Table.

Although the cited references use hash tables to solve certain problems, we found few works regarding this particular approach, as most references to hash tables used in trees correspond to hash trees, in the sense that the nodes of a tree are hashes of certain data sets; in contrast our work proposes the opposite: to build a hash table from a tree. We are focused on building a hash table assuming that traversing several times the same nodes is mostly useful when the space available (memory or otherwise) is restricted; otherwise, this method takes advantage of the fact that when enough space is available, it is useful to cache certain data so as to reduce the number of times each node is visited.

Our work provides a novel idea for data interpretation stored in trees, since we show that approaching trees in a different manner can transform their disadvantages into key strengths, at the same time making their advantages into clear weaknesses. Therefore, we provide a way to choose between different tree performance behaviors, depending on the problem to be solved, thus providing the ability to optimize certain features when necessary.

3. Algorithm

In this section we describe the proposed algorithm for the transformation of any rooted tree, into an H-Tree, comprised of a hash table containing pairs of the form <N, P> (or <P,N> for cases such as Huffman coding), where N is a node of the tree and P it's corresponding path from the root. The advantage of this algorithm is that it will return the path to any node in at most two lookup operations on the hash table, at the cost of more memory usage. The basic flow of the algorithm is shown in the following figure:

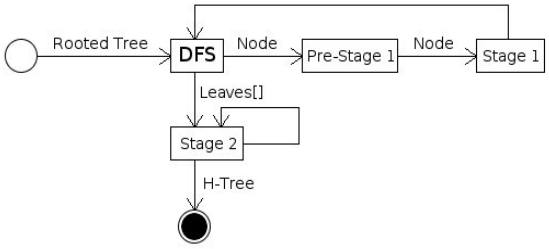

Figure 1. Algorithm flow.

The algorithm receives a rooted tree, which is then traversed in a Depth-First Search fashion. The block named "Pre-Stage 1" receives each node visited by the DFS block, whose task is to make the path point at the parent of the current node. Next in line is the block "Stage 1", taking the node from the previous block and checking whether the node is a leaf or not, adding it to the hash table in the former case, or to the path in the latter. When all leaves are visited, an array is sent to "Stage 2" which is concerned with associating each of the remaining nodes (non-leaves) to their corresponding leaf (that which contains the node in it's path to the root), and adding such pair to the hash table; the result is then an H-Tree.

3.1 DFS block

This block is made up of a regular Depth-First Search algorithm used to traverse the tree, processing each visited node on the Pre-Stage1 and Stage1 blocks.

3.2 Pre-Stage 1 block

This block receives a node N, processing it in the following way:

```
Pre-Stage1(N){
1    if (N.depth > Current_depth){
2      Stage1(N);
3    }
4    else if (N.depth == Current_depth){
5      Path.removeLast();
6      Stage1(N);
7    }
8    else if (N.depth < Current_depth ) {
9      Path.removeNodes(Current_depth-N.depth+1);
10     Stage1(N);
11   }
12 }
```

Note that *Current_depth* is a variable used to keep track of the depth from the last visited internal

node. This variable is needed in line 9, to indicate that the last *Current_depth* nodes should be removed, guaranteeing that the path is actually pointing to the parent of node *N*.

3.3 Stage 1 block

```
Stage1(N){
1    if ( isLeaf(N) == false){
2        Path.add(N);
3        Current_depth = N.depth;
4    }
5    else {
6        HashTable.add(N, Path);
7        Leaves[].add(N);
8        }
9  }
```

3.4 Stage 2 block

This stage receives an array comprised of all the leaves in the rooted tree, checking each leave separately:

```
Stage2(Leaves[]){
1          for (i = 0; i < Leaves.size; i++){
2             Path  = HashTable.get(Leaves[i]);
3             for (j = Path.size; j >= 0; j--){
4                HashTable.add(Path.node[j], Leaves[i]);
5             }
6          }
7  }
```

4. Analysis

The H-Tree structure is generated by first finding the path from the root to each leaf and storing this information on a preliminary H-Tree; then, each internal node I is paired up with that leaf, whose path to the root includes precisely I.

This way, the original rooted tree will be traversed only once in a Depth-First fashion, and all the corresponding paths from the root to each node will have been calculated, needing minimal lookup time and at most two lookup operations over the table. As such, it can be pretty easily noted that the main idea is to switch from a computationally-intensive approach to a more memory-intensive method.

Speaking in general terms, the preprocessing complexity of the algorithm can be stated as $O(H\text{-}Tree) = O(DFS) + O(PS1) + O(S1) + O(S2)$. Where $O(DFS)=O(|V|+|E|)$ [4], $O(PS1)$ is the Pre-Stage1 block complexity, and $O(S1)$ and $O(S2)$ the orders of Stage1 and Stage 2 accordingly. Breaking up each factor, we get the following:

For Pre-Stage1 it is fairly obvious that $O(PS1) = 2+(Depth*Path.remove) = 2+(Depth*O(1)) = O(D)$ where D is the maximum depth of the tree; at Stage1 we have $O(S1) = O(1)+O(1)+O(1)=O(1)$. Finally, for Stage2 we can see that there are two nested FOR loops, depending on the number of leaves L and the size of the path which is equivalent to the depth, such that we get $O(S2)=L*D$.

Finally, we obtain the result:

$O(|V|+|E|) + O(D) + O(L*D) = O(|V|+|E|) + O(L*D)$.

The worst case of this approach would be either one of two cases: having a tree with the structure of a list, in which case we would only have a single leaf of depth D, and thus D entries on the hash table, with the first one corresponding to that single leaf, and the $D-1$ remaining ones to each of the internal nodes; the search for the last internal node would imply two lookup operations plus a sequential search over $D-1$ elements, as in Figure 2.

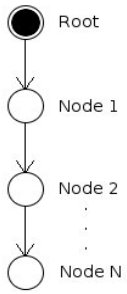

Figure 2. Tree with a list structure.

The worst case scenario can also be seen as a tree with C nodes, all children of the root, shown in Figure 3, in which case the generated H-Tree would have C entries with the exact same path to the root, thus needing only one lookup operation per path search (as every node is a leaf) but having C times more storage space than actually needed.

Also, we have to note that the heaviest running time load comes from the Stage2 block, which can be avoided for problems that won't have any use for internal nodes, such as the proposed Huffman coding optimization. In a case like that, we would only need to process the tree up to Stage1, obtaining a running time of $O(|V|+|E|) + O(D)$. On the other hand, the query complexity will be $O(1)$, resulting from the use of a hash table [4].

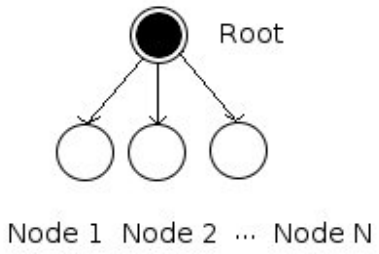

Figure 3. Tree of depth one.

5. Conclusions and results

In this section we present various results from experiments conducted on our own implementation of the algorithm. We coded different text files from The Canterbury Corpus benchmark file set [5], and then proceeded to decode the resulting files using an H-Tree data structure, counting the number of hash table lookup operations needed.

The dataset consisted on the following text files:

Table 1. Input data

File	Description
Alice29.txt	English text
Lcet10.txt	Technical writing
Alphabet.txt	Repetitions of the alphabet
E.coli	Complete genome of the E. Coli bacterium
Bible.txt	The King James version of the bible
World192.txt	The CIA world fact book
Pi.txt	The first million digits of pi

We processed each file separately, counting the character frequency so as to build the Huffman code tree; the compressed file was then decoded using the H-Tree method proposed, by reading the first *m* bits (where *m* is the length of the shortest codeword) and using them as a lookup key; if the code was not found, the next bit would be appended and another lookup was performed, repeating these steps until a valid codeword was found, just to start again with the next *m* bits. The results are shown in table 2, where the column "Characters" displays the number of characters in the original file and "Compressed" the number of bits after coding it.

In table 3 we present the minimum and maximum codeword lengths, which correspond to the depths of the shallowest and deepest leaf nodes accordingly. These parameters are important, since the number of lookup operations needed to decode each file will depend on the shortest and longest codeword lengths, as well as the number of characters that need to be decoded.

Table 2. Processed data

Text file	Characters	Compressed
alice29.txt	144873	651441
lcet10.txt	411716	1896058
bible.txt	4017009	17487924
world192.txt	2343162	11600971
e.coli	46386902	9277380
alphabet.txt	100000	476920
pi.txt	100000	3399064

Table 3. Codeword data

Text file	Shortest code	Longest code
alice29.txt	2	17
lcet10.txt	3	17
bible.txt	2	17
world192.txt	3	21
e.coli	2	2
alphabet.txt	4	5
pi.txt	3	4

The best case scenario will be the one where the length of all the codewords in the bit stream, are equal to *m*, resulting in *n* lookups, where *n* is the number of coded characters in the compressed file, since we would only need one lookup per codeword.

On the other hand the worst case would be a file where all the characters have a codeword length equal to *M* (the length of the longest codeword) with $m<M$, resulting in $n((M-m)+1)$ lookups.

In contrast, if we where to traverse a Huffman tree using the bit stream as a decision parameter to select the left or right node accordingly, we would have at least $n(m+1)$ operations for the same best case scenario as above, since we need *m* left-right choices plus one array access to get the corresponding symbol, times *n* characters. Similarly, on the same worst case scenario as previously supposed, we have $n(M+1)$ operations for the same reasons.

Table 4 shows the data corresponding to the number of operations needed for decoding each file using a tree structure and an H-Tree, as well as the lookup reduction ratio. As we can see, the number is lower if we use an H-Tree, ranging from 1.57 to 12 times less in the case of the genome for the E.Coli bacterium.

The three files with the highest ratios correspond to those for which the difference between *M* and *m* is smaller, in accordance to the analysis presented previously. We can see in table 5, that for the e.coli text file, with a $D-d$ value of 0, we get the highest lookup reduction since we have a best case

scenario where all codewords are equal in length, needing only *n* operations.

Table 4. Lookup data

Text file	Tree	H-Tree	Ratio
alice29.txt	796314	506568	1.57
lcet10.txt	2307774	1072626	2.15
bible.txt	21504933	13470915	1.6
world192.txt	13944133	6914647	2.02
e.coli	55664282	4638690	12
alphabet.txt	576920	176920	3.26
pi.txt	3499064	1399064	2.5

Table 5. Length difference and reduction ratio

Text file	Ratio	Difference
e.coli	12	0
alphabet.txt	3.26	1
pi.txt	2.5	1
lcet10.txt	2.15	14
world192.txt	2.02	18
bible.txt	1.6	15
alice29.txt	1.57	15

On those files where the difference is big, the resulting ratio is smaller, although we need to take into account the *m* and *M* values and not only their differences, such as the world192.txt file case, where the difference is bigger than bible.txt and alice29.txt.

Table 6. Code lengths

Text file	Ratio	Shortest code	Longest code
e.coli	12	2	2
alphabet.txt	3.26	4	5
pi.txt	2.5	3	4
lcet10.txt	2.15	3	17
world192.txt	2.02	3	21
bible.txt	1.6	2	17
alice29.txt	1.57	2	17

The analysis for the average case is not presented, since a good file set of "average" text files would be too hard to compile. The number of possible symbols, different *D-d* values and Huffman code tree structres are all parameters that need to be carefully selected to get a good approximation.

Although, from what we can observe in the experimental data, we can say that using this method can result in dramatic improvements on the number of lookup operations specially when a good data analysis is conducted, particularly useful when cpu time requirements are restricted.

6. Future work and improvements

In this section we present several improvements that can be included in the algorithm, specially optimization ones. As it is, the algorithm could use a more precise way of building the H-Tree, for example by first checking whether the node being added to the hash table has the same path as some other nodes already in it, providing those with the same parent, the possibility of clustering, allowing for some space saving.

Also, a threshold would be very helpful on balancing the cpu-memory requirements. If only some subtrees are to be selected by some criteria off the main rooted tree to be processed, several H-Trees could be generated corresponding to each of the subtrees, improving the cpu-memory needs.

The most important improvement in our future line of research is to add the ability for the algorithm to select the optimal or near-optimal threshold for the aforementioned balance for different types of rooted trees, so an analysis stage would be set before Pre-Stage 1, to determine if the process should aim for a balanced binary tree, B-Tree, k-d tree, etc. and maybe even detecting the amount of available memory and adapt accordingly.

7. References

[1] M. A. Hai Zahid, Ankush Mittal, R. C. Joshi, "Least Common Ancestor Based Method for Efficiently Constructing Rooted Supertrees", Journal of Bioinformatics and Medical Engineering (BIME), The International Congress for Global Science and Technology (ICGST), pp 3-4.

[2] Sairam Subramanian, Sridhar Ramaswamy, "The P-Range Tree: A New Data Structure for Range Checking in Secondary Memory"

[3] Sriram Ramabhadran, Joseph M. Hellerstein, Sylvia Ratnasamy, Scott Shenker, "Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables"

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to algorithms", First edition, MIT Press and McGraw-Hill, 1995, ISBN 0-262-03141-8 (MIT Press), ISBN 0-07-013143-0 (McGraw-Hill).

[5] Tim Bell, Matt Powell, "The Canterbury Corpus"