EE 274 Project Report - Adaptive Huffman Codes

Sudeep Narala
Raymond Yang

December 2022

Abstract

Huffman codes require two passes through the data, first constructing the source distribution and then encoding the data. The decoder requires this source distribution to decode. Adaptive Huffman codes require only a single pass through the data and the decoder does not require knowledge of the source distribution at all. In this project, we reviewed adaptive Huffman codes realized with Vitter's algorithm and implemented adaptive Huffman codes using the circularly-linked-arrays data structure proposed by Vitter and a tree-based data structure. Nonetheless, our implementation follows the same codebook-update procedure as proposed by Vitter.

1 Introduction

Huffman coding is an optimal prefix-free code used for lossless data compression. Huffman coding uses a two-pass procedure to encode data. In the first pass, the empirical source distribution is computed and represented by the Huffman tree. In the second pass, the data are encoded based on a codebook derived from this Huffman tree. When decoding, the decoder must know the source distribution in addition to the encoded data.

For our project, we explored a variant of Huffman coding —adaptive Huffman codes. This adaptive variant of Huffman codes eliminates the need for a known source distribution when encoding and decoding data. In adaptive Huffman coding, only a single pass is needed, where the source distribution and the codebook are updated on-the-fly as data is seen and encoded by the encoder. One algorithm proposed for this task is Jeffrey Scott Vitter's algorithm (Vitter's algorithm), which uses a circularly-linked-arrays data structure. We will refer to this as Vitter's implementation. Vitter claims this algorithm to use 1 less bit on average compared to two-pass Huffman codes [2]. In our project, we implemented this algorithm using both Vitter's implementation [3] as well as a more intuitive binary tree-based data structure, which we will refer to as the naive breadth-first search (BFS) implementation. The following sections will describe existing implementations of adaptive Huffman coding, our methodology and results.

2 Prior Work

(This section is mostly borrowed, with some modifications, from the midpoint review because our prior work hasn't changed since then.)

Our main prior work was both of Vitter's papers. Obviously, all this work relies on Huffman's work [1] in creating the optimal prefix-free codes. The first paper is a discussion of Vitter's Algorithm and its optimality for an adaptive huffman codec. The second paper is the implementation of Vitter's Algorithm with very good runtime properties. In the first paper, Design and Analysis of Dynamic Huffman Codes, Vitter presents his single-pass huffman encoding algorithm. The goal of the algorithm is to construct a huffman tree on the fly where the tree has the properties as if it was constructed using the empirical distribution created using all the data we have seen so far. One intuition Vitter proposes for why his algorithm is optimal is that he optimizes for not just expected code length (which any valid Huffman tree would accomplish) but also average unweighted length and max length. This helps guarantee that the next unseen symbol is encoded with a length that is not too large. To briefly summarize the actual algorithm, Vitter's algorithm involves mutating the tree (i.e. changing its structure) such that when the leaf node corresponding to the symbol we just saw is incremented by 1, and all internal nodes in the path to root are consequently incremented by 1, the tree still satisfies the property of being a Huffman tree. In order to accomplish this, there is a function which performs a slide and increment which does a bulk of the heavy lifting in the algorithm. This function moves the node to the right in the mutated tree and increments its value (1 execution of the function is done all the way to root).

In the second paper, Dynamic Huffman Coding, Vitter gets in the weeds about how to go about implementing the algorithm optimally. He presents the idea of a block, where each block contains nodes which are of the same type (internal or leaf) and weight. Each block has a leader which is the highest node in the block in terms of implicit numbering (nodes numbered ascending from bottom-to-top, and left-to-right within each level). He introduces what his data structure needs to be able to accomplish: represent a huffman tree, store blocks and their leaders, interchange leaves, be able to slide a node by increasing its weight by 1, store correspondence between leaves and letters of alphabet. The rest of the paper is the actual implementation and primitive data structures used to materialize the presented data structure's properties. However, the code presented in the paper is in Scala and was a little bit hard to read and digest. The code is also light on documentation, which we attempted to, and are continuing to improve in our Python version.

3 Method

Existing algorithms for depth-limited Huffman codes and adaptive Huffman codes are available. We will first implement an algorithm for depth-limited Huffman codes and also Vitter's algorithm for adaptive Huffman codes in Python as proof-of-concept. With a functional Python implementation, we plan to use Rust to improve the speed of the algorithm while moving towards an age of memory safe compressors.

3.1 Algorithm Details

In the below, NYT refers to "Not Yet Transferred", i.e. an alphabet element that hasn't been seen yet.

```
update:
    if NYT:
        create new internal and leaf node (sibling of NYT)
        leaf_to_increment = new leaf node
        p = parent of leaf node (new internal node)
    else:
        make leaf node (corresponding to alphabet element) leader
        p = the leaf node
    while p is not NULL (i.e. above root):
        slide_and_increment(p)
    if leaf_to_increment:
        (the reason for this special treatment is because if you started
        at the new leaf node, you would potentially attempt to slide
        above the new internal node created which doesn't make much sense)
        slide_and_increment(p)
slide_and_increment(p):
    if p is internal:
        increase p's weight by 1
        slide higher than all leaf nodes which have same weight
        (the reason we use old parent is because if p does indeed slide,
        i.e. a leaf node with higher weight than p's previous weight went left,
        the old side is the one that needs incrementing)
        (this could still be the same parent if no sliding!)
        p = old parent of p
    else:
        increase p's weight by 1
        slide p higher than all internal blocks of lower weight than p
        p = parent of p
```

When we refer to an element sliding, we mean it moves higher in implicit numbering. Implicit number increases bottom-to-top, left-to-right. We show an example below to solidify this idea.

3.2 Example: "test"

Encoding the string "test" as an example, the evolution of the tree as each input is processed is shown in Figure 1. Here our test input is "test" processed sequentially. Each row corresponds to the update of the tree with a new character input. The input is shown on the left side and the arrows indicate the evolution of the tree. The meaning of different shapes and numbers are described in the legend on the bottom right side. The implicit ordering is basically a numbering of the nodes such that, by ordering the nodes in such way, the invariant property is maintained. The invariant property in Vitter's algorithm necessitates that nodes with smaller weight always precede nodes with larger weight, and leaves with the same weight always precede internal nodes of the same weight. This property is enforced so that the update procedure is canonical. The implicit order is smallest for the node in the lower left corner. The order increases horizontally first before moving up a level and continuing from the leftmost node in the new level. The root node always has the largest implicit order.

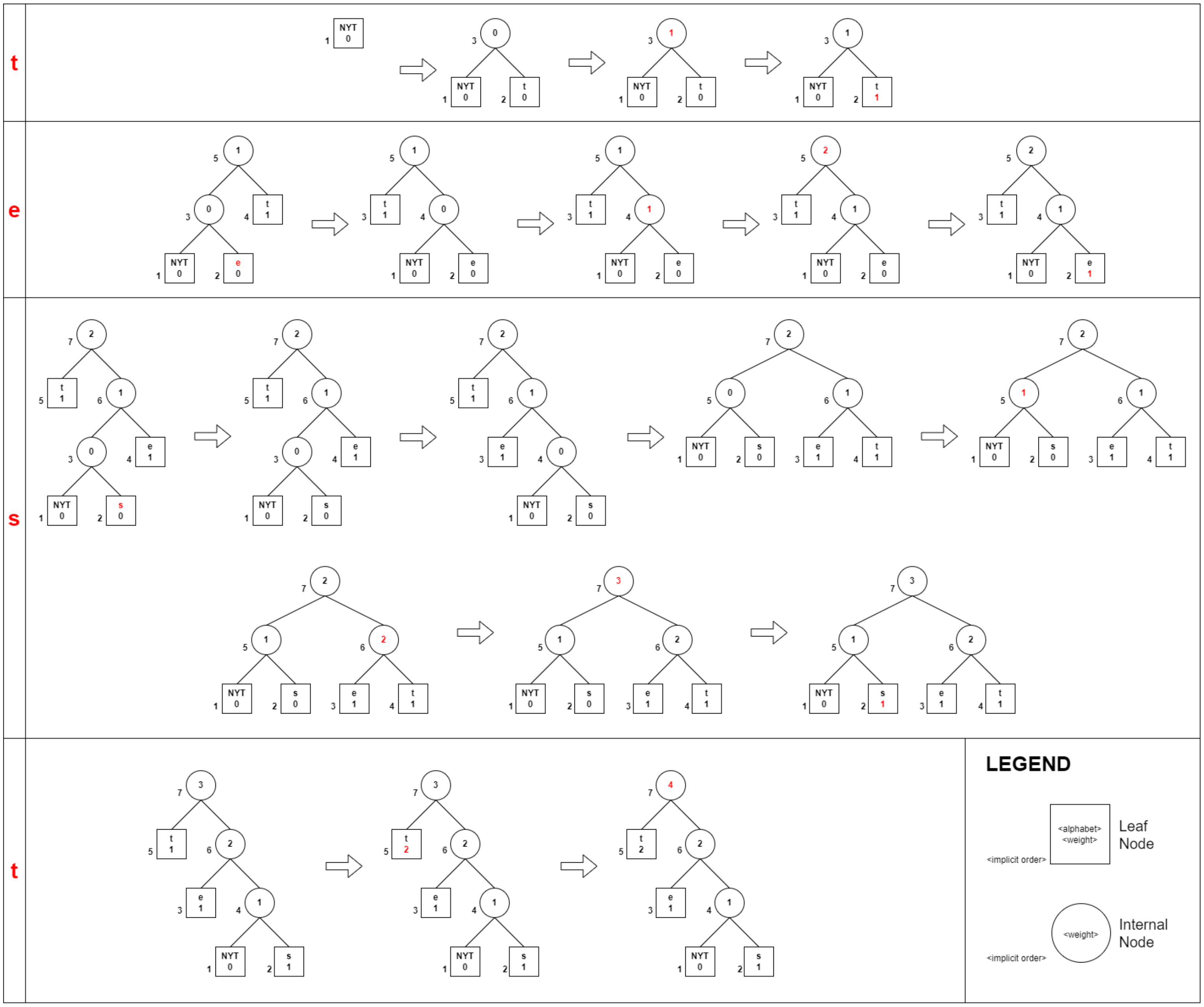

Figure 1: Tree evolution for example ”test”

3.3 Implementations

3.3.1 Naive BFS Implementation

The naive BFS implementation adopts a binary tree-based data structure, as mentioned before. When a new symbol occurs, we adopt the same logic shown in the pseudo-code and create 2 new nodes: one internal and one leaf. When slide and increment is called, we swap nodes one at a time until the node reaches the right place. Thus, we literally slide it over. In order to do this, we maintain a parent pointer which allows us to modify pointers appropriately. The sliding stops when the node either reaches its appropriate post-incremented weight block (i.e. alongside, in implicit numbering, other leaf or internal nodes of the same weight) or it gets into a block of its own where it is appropriately wedged between 2 other blocks. Ex: a leaf node of weight 5 ends up between 2 leaf nodes of weight 4 (on the lower implicit number side) and weight 6 (on the higher implicit number side). Finally, each time the slide and increment is completed, we update the implicit numbers of every node in BFS fashion. I.e. the root node starts at some high number and we count down from there. We perform BFS in a right to left manner such that nodes in the same level start getting numbered from the right and continue towards the left (decrementing the numbering each time). We maintain a pointer from this ”implicit” number assigned and nodes. We also maintain a pointer from alphabet element (i.e. ’a’, ’b’ etc.) to node and node to assigned implicit number so we can determine where a node needs to slide to (i.e. swap with the next implicit number).

3.3.2 Optimized Vitter's Implementation

To the best of our knowledge, this is the first attempt at an open-source accurate-to-the-original implementation of Vitter's Algorithm in Python (making it much more readable). We did our best to document the code in-line, but there is definitely a little more work to do. While it is hard to explain all aspects of Vitter's Algorithm, we will go over some high-level details. We refer you to the code for more details.

For the discussion of this algorithm, we will assume one is trying to encode with 8 bit block lengths (thus the total alphabet size is 256). We also assume 0-indexing as is done in Python and not 1-indexing as is done in Scala. Some discrepancies from the original paper arise because of this in our implementation and the following discussion.

In order to maintain the same invariants, Vitter made use of many arrays which kept track of the tree structure. He refers to his tree as a "floating tree" because it is not stored the traditional way with left and right child pointers. Explicit number starts at 255 and goes downwards for leaf nodes and starts at 510 and fills downwards for internal nodes (510 is chosen because this is the total number of internal nodes possible if all elements of the alphabet are seen) as new nodes are created. Each of the leaf node numbers corresponds to an element in the alphabet. The NYT node is represented by 255 initially and counts down for every additional alphabet element seen.

The fundamental unit of organization is a "block". In the algorithm, nodes are numbered explicitly (as opposed to the implicit numbering discussed earlier). A block consists of nodes of the same weight and same type. All nodes in one block are either all leaves or all internal nodes. Every node knows what block it is a part of. Additionally, the parity (i.e. whether the leader is a left child or right child), leader node and last node (i.e. smallest in explicit/implicit numbering) of a block are maintained. Additionally, since we need to know the ordering of blocks for the sliding process, a circular linked list is maintained between the blocks with pointers to the next and previous block from the current one. The reason blocks are chosen as a unit is because sliding happens across a whole block when it does occur in slide and increment (as shown in the pseudocode earlier).

Now, let's go over the actual operation of the algorithm. Every time a new node is seen, 2 new nodes are created: a leaf node for the new element and an internal node which is the parent of NYT and this new leaf node. This also creates a new block for the internal node which is a block of internal nodes of weight 0. The new node is added to the same block as the NYT node and becomes the new leader. Next, the slide and increment procedure starts from the parent of the new node. If the node is a leaf node, it needs to slide ahead of all internal nodes of the weight it is (because after the increment, it will be higher). In order to do this, we check whether the next block (which we have access to because of the circularly linked list) is an internal node block with weight equal to the current leaf weight. If this is the case, Vitter uses node offset calculations in order to find the location for the node. In a nutshell, since Vitter maintains the leaders location, he is able to find the right place. A similar process ensues for internal nodes. It is important to note that this process can also cause the creation of a new block if no block is present with the weight the node is about to be after incrementing and node type. Finally, slide and increment takes place for the new node itself (which actually just involves incrementing its weight as no movement takes place).

For a node that isn't new, the process is very similar. As discussed earlier, the slide and increment starts at the target node and works its way up to root.

Some interesting parts of the algorithm to us were how Vitter traverses up (for encoding) and down (for decoding) the tree. When he traverses up the tree (to find out where the target leaf node is relative to root), he makes very clever and tricky use of the leader and parent pointers, along with the parity of blocks, that he maintains accurately throughout. A key invariant that he utilizes here is that every node (other than root) has a sibling under his construction (and any Huffman tree). He utilizes the fact that in order to get from the parent of a nodes block to the parent of the current node, it takes half the number of steps as it takes (approximately, this is where parity comes into to play) to get from the leader of the block to the node (in explicit/implicit numbering). This is how he is able to pin-point a node's parent. In order to find the child of a node, he takes advantage of his maintenance of leader, right child and last node pointers of a block. He is then able to do simple offset calculations if the right child of the leader is in the same block as the node's child. Otherwise, he is able to go to the previous block and do a similar calculation (glossing over some details here).

Overall, the algorithm ended up being extremely complex and involving a lot of in the weeds pointer arithmetic. However, it was also beautiful that he was able to construct a tree that was optimized for what he was trying to accomplish, and as we will see, ended up being extremely efficient.

4 Results

We ran experiments on 2 different file sizes with different numbers of unique elements in the alphabet with our naive BFS implementation and an optimized version of Vitter's written in Python. The results are shown in the above figure.

Unfortunately, we were only able to run with such small alphabet sizes because our optimized Vitter's algorithm implementation had a bug which activates at larger file size (with many unique elements in the alphabet).

Even from the experiments run, we can see that the optimized implementation does a lot better.

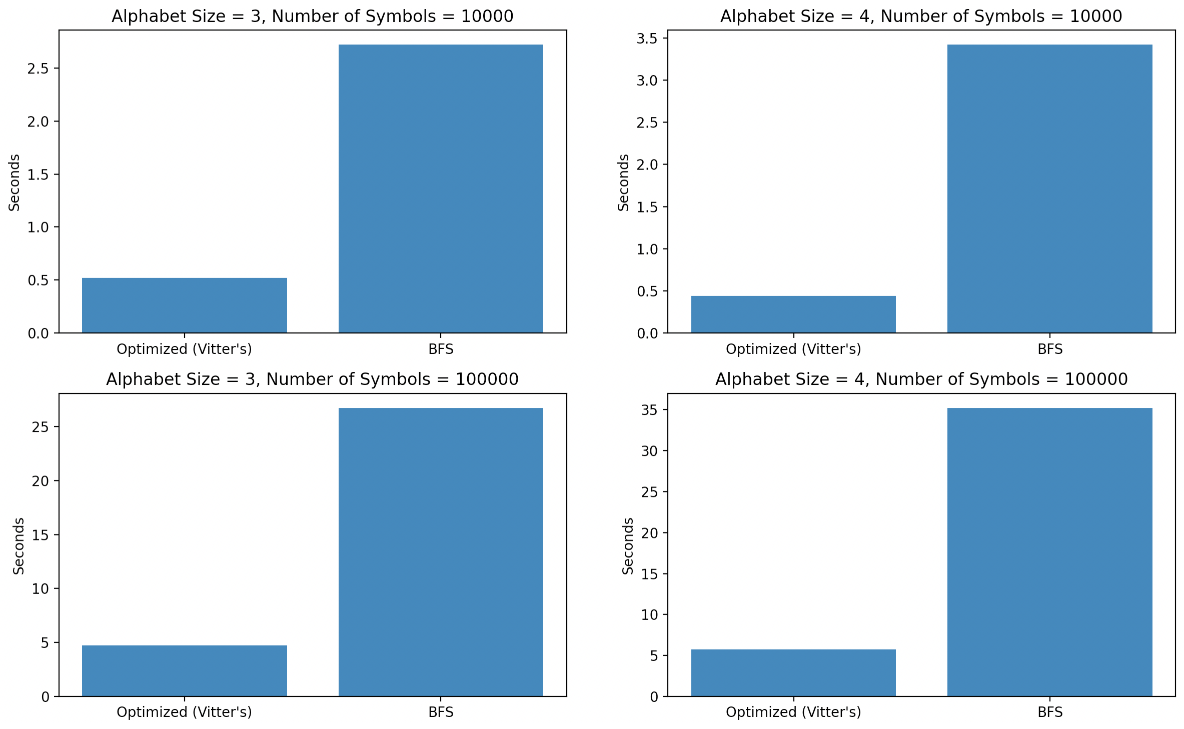

Figure 2: Benchmark between naive BFS Method and Optimized Vitter's Implementation

5 Conclusion

During this process, we gained a lot of insight into how difficult it is to implement a complex codec. Initially, we thought the task of writing the codec in Python would be simple enough that we could move on to Rust. However, as noted earlier, we still have minor bugs in our Python code. The problem with debugging the code dynamically (i.e. at runtime) is that it was very difficult to determine where the error occurred. For example, even though an out of bounds error could occur while encoding a certain symbol, that was often the result of some metadata being incorrectly mutated a few symbols ago. However, going through this debugging ensured that we understood Vitter's implementation inside and out. It was very interesting to see all the considerations which went into the algorithm and it is very useful for our future algorithm design skills.

References

[1] D.A. Huffman. A method for the construction of minimum redundancy codes, June 1951.

[2] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes - i2s, Oct 1987.

[3] Jeffrey Scott Vitter. Dynamic huffman coding, June 1989.