Culminating Deep Dive On Trees: Huffman Encoding and Tic-Tac-Toe

Jules Brettle, Charlotte Ramiro, Florian Schwarzinger, Jane Sieving

February 14, 2022

1 Huffman Coding

One application of trees is in lossless data compression using Huffman codes. Huffman coding is a type of prefix code that uses binary trees in order to efficiently encode messages. Prefix codes (also known as prefix-free binary codes) have the property that no whole codeword is the prefix for any other codeword. This enables unambiguous parsing of a variable-length code.

Another key property of Huffman coding is that the length of the codeword for a given symbol is inversely proportional to the frequency of the symbol. As a result, symbols are optimally mapped to codewords to minimize the length of the encoded message.

We created an implementation of the Huffman coding scheme in Python that would both encode and decode a message. We used a technique called canonical Huffman coding to enable decoding of message while limiting the amount of data required to communicate the codeword mappings.

1.1 General Huffman Coding

Huffman coding first requires generating the prefix code for the specific text to be encoded. This involves constructing a binary tree with branches representing a bit in the codeword and common symbols near the root. First, leaf nodes are created for each symbol and added to a priority queue from lowest to highest frequency. Then, the two highest-priority nodes are removed and a new internal node is created with these two nodes as children and a total frequency equal to the sum of the children's frequencies. This node is then enqueued, and the process continues until there is only one node, the root, remaining in the queue (which will represent an empty codeword and have all symbols as descendants). A visual representation of the binary tree creation for a short 6-symbol message is shown in Figure 1.

Once the tree is constructed, following a path from root to leaf and appending 0 for a left child and 1 for a right child will yield the code word for the symbol at the leaf.

Since the encoding generated by the tree is dependent on the frequency of symbols in the original message, it is necessary for the recipient to be able to reconstruct the encoding. There are several methods to enable this, and we chose to explore prepending a fixed-length encoding of the canonical Huffman code corresponding to the derived code.

1.2 Canonical Huffman Coding

Often, to generate a canonical Huffman code, the normal Huffman encoding is found first and converted to a canonical one. We will consider the canonical Huffman code for the message "A_DEAD_DAD_CEDED_A_BAD_BABE_A_BEADED_ABACA_BED".

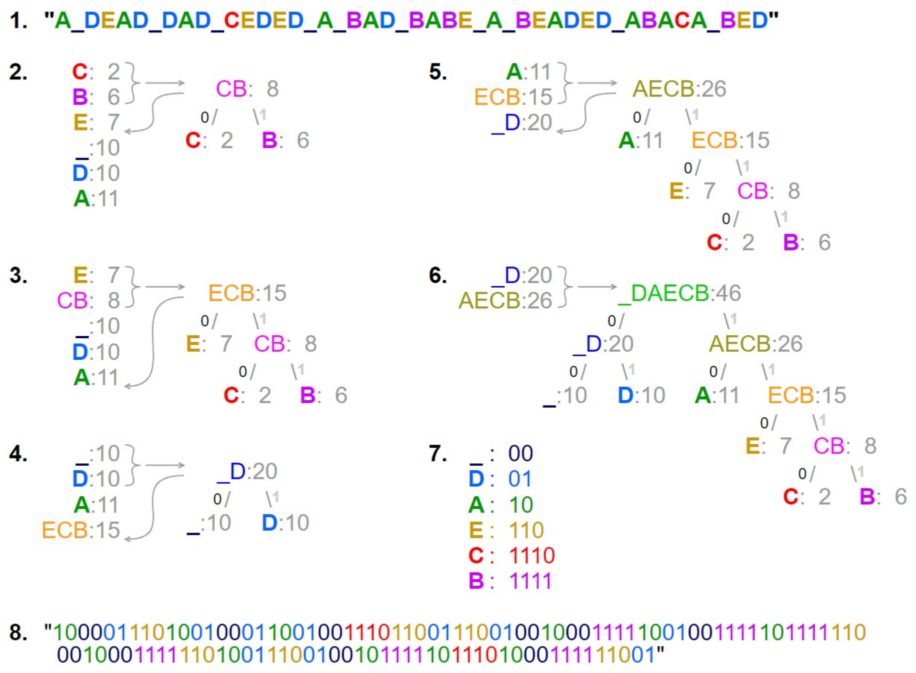

Figure 1: A visualization of the Huffman algorithm its binary tree. This image was taken from the Wikipedia page for Huffman codes.

Table 1 shows the normal Huffman codes for the characters, listed from highest to lowest frequencies.

Character	Code Word
_	00
D	01
A	10
E	110
C	1110
B	1111

Table 1: A table of the normal Huffman codes for our sample message.

In order to canonize the normal Huffman code, the original symbol-codeword mapping is sorted primarily by codeword length and secondarily by alphabetical order of symbol, creating Table 2. Note that the underscore symbol ranks lower alphabetically than capital letters as per Python sorting.

Next, each codeword is replaced with one of its same length using the following method:

- The first symbol in the list gets a codeword of its same length but all zeroes.
- Each subsequent symbol is assigned the next binary number in sequence, thus ensuring that the following codes are higher in value.
- When a longer codeword is reached, after incrementing the binary value, zeroes are appended until the codeword has the same length as the original codeword (also known as a left shift).

This algorithm produces the codebook in Table 3.

Character	Code Word
A	10
D	01
_	00
E	110
B	1111
C	1110

Table 2: A table of the normal Huffman codes for our sample string ordered such that we can convert them to canonical codewords.

Character	Code Word
A	00
D	01
_	10
E	110
B	1110
C	1111

Table 3: A table of the canonical Huffman codewords for our sample string.

1.3 Decoding

We decided to use canonical codes because simplifies the decoding process by standardizing the symbol mapping. In order to decode normal Huffman codes, the decoder needs to re-generate the binary tree from the frequency count of each character, which requires sending a lot of metadata along with the encoded message. With canonical codes, the message can be decoded with far less bytes because so much about the codeword assignment is standardized. By transmitting just the length of all codewords, the symbol mapping can be reconstructed.

We chose this method with the decoding process described below because for an 8-bit symbol set, exactly 145 bytes are needed to encode the symbol mapping, regardless of the number of symbols used. This is in contrast to an encoding which transmits specific characters and data about their positions in the tree, which would require a minimum of 256 bytes just to encode the symbols if all symbols were used.

To enable decoding of the message, two pieces of data must be created: `min_cw[ ]` and `cw_to_char[ ][ ]`. `min_cw[i]` represents the numerical value of the minimum codeword of length `i`, while `cw_to_char[i][j]` shows the `j`th symbol with a codeword of length `i`. For example, `cw_to_char[2][1]` points to D, the 1th index among codewords of length 2. A table representing this data is shown in Table 4.

Length (i)	min_cw[i]	cw_to_char[i][ ]
2	0	A, D, _
3	6	E
4	14	B, C

Table 4: A table of the necessary information for canonical decoding.

This table can be encoded in a known-length bitstring of `sum_through(1, n) + log2(n) *` `a` bits, where `n` is the maximum length of any codeword and `a` is the number of symbols in the alphabet being used.

Now, to decode, the encoded bitstring is read and appended one bit at a time to a buffer `cw_buffer` until a codeword is completed. Then, the codeword is decoded, the resulting symbol is appended to toe output, and `cw_buffer` is cleared. This repeats until the full string has been decoded. Since the minimum value of the codeword for any length is known, the numerical value of the current buffer is compared to the minimum codeword of that length. If it is less than the minimum codeword, another bit is appended to the buffer and the process continues. If it is at least the minumum codeword value, then the current buffer *could* be a codeword, so it is searched for in `cw_to_char[len(x)][j]` where `j` is the numerical value of `cw_buffer` minus the numerical value of the smallest codeword of length `i`. The pseudocode for this is written below for clarity, but the actual code can be found in the appendix.

```
if numerical_value(x) >= min_cw[len(x)] :
    buffer may be codeword
    find codeword in cw_to_char[len(x)][numerical_value(x) - min_cw(len(x))]
else:
    append next bit in encoded bitstring to cw_buffer and re-check for isCodeWord
```

In this way, the entire original message can be reconstructed while prepending a limited amount of data to convey the encoding.

Using our program, we tried encoding and recovering both one of our heavily-used online sources for this work, and the Python code itself. The Scranton Canonical Huffman Coding page was reduced from 27.8 to 17.8 KB, a 36% reduction. `huffman.py` was reduced from 5.65 to 3.52 KB, a 38% reduction. When decoded, the former rendered as an identical web page, and the latter could be run again on itself.

2 Minimax Algorithm With Tic-Tac-Toe Trees

2.1 What is Minimax?

The Minimax algorithm is an algorithm that is typically used to find optimal moves in a 2-player, turn based game such as Chess, Connect-Four, Mancala, or tic-tac-toe. In this algorithm, every game/board state has a value associated with it which represents which player the current board state benefits. The larger the number, the more it benefits one player and the smaller the number, the more it benefits the other player. Because of this, one player is always trying to make the moves that would maximize the value and the other is trying to minimize the value. These players are known as the maximizer and minimizer respectively. In most applications, the value is centered around 0 so values greater than 0 imply the maximizer is favored and values less than 0 imply the minimizer is favored. These values are generated by some heuristic that varies from game to game.

To use the Minimax algorithm, your graph needs to be set up as a tree where the nodes represent game states and each edge represents the singular move needed to get from the parent state to the child state. In order to generate values for each node, you want to recursively take either the maximum (if it's the maximizer's turn at the node you are at) or the minimum (if it's the minimizer's turn to act) value of the children of the node you are looking at. The base case of this, when a node doesn't have any children nodes, is where the heuristic comes into play. The heuristic is applied to the leaf nodes and the resulting values are propagated back up the tree as appropriate. The resulting tree allows for each move to have a value associated with it and makes it very easy for both the maximizer and minimizer to find their optimal move to play.

2.2 Using Minimax for Tic-Tac-Toe

The Minimax algorithm, as described above, can be used to find the best moves in a tic-tac-toe game, and create an unbeatable AI.

We can start by giving each ending board a value based on whether it shows a tie, win, or loss for X. A win for us (X) when there are no blanks remaining (such as in Board 9 in Figure 2) we can give the value of 1. A tie would receive a value of 0. Any situation where the opponent (O) wins we would value as negative. To encourage the algorithm to choose paths that lead to quicker wins, we add a point for each blank space at the win and subtract 1 for each blank space at a loss. For example, Board 3, where O wins with 1 space remaining, is valued at a -2, and Board 5, where X wins with 2 spaces remaining, is valued at a 3. We then pass these final values up the tree to give each decision a weight.

We assume that O will choose the minimum weight decision and X should choose the maximum weight decision. If the opponent encounters Figure 2's Board 2, they have the choice between the move that creates Board 3, weighted -2, or the move that creates Board 4, weighted 0. They should choose the minimum weight (-2) and win the game. Similarly, if they encounter Board 7, they could move to create Board 8, weighted 1, or Board 10, weighted 0. They should choose the move of minimum weight (0) and tie rather than lose.

If X encounters Board 1 (from Figure 2), they have 3 choices for where to place their next marker. These are represented by the red arrows, with the weights for each choice in the grey boxes beside each arrow. Moving to create Board 2 is weighted -2, because it allows O to win on the next turn. Moving to create Board 6 is weighted 3, because it instantly awards a win to X, with 2 blank spaces left. Moving to create Board 7 is weighted 0, because although it is possible that X would still win, the best move for O would force a tie. X should choose the maximum (3) and win the game. This also means that the weight of the decision that leads to Board 1 is 3.

The weights would propagate up the whole tic-tac-toe tree as explained above, giving a computer program with this algorithm a clear way to assess the goodness of each possible move and make the best move for any game state.

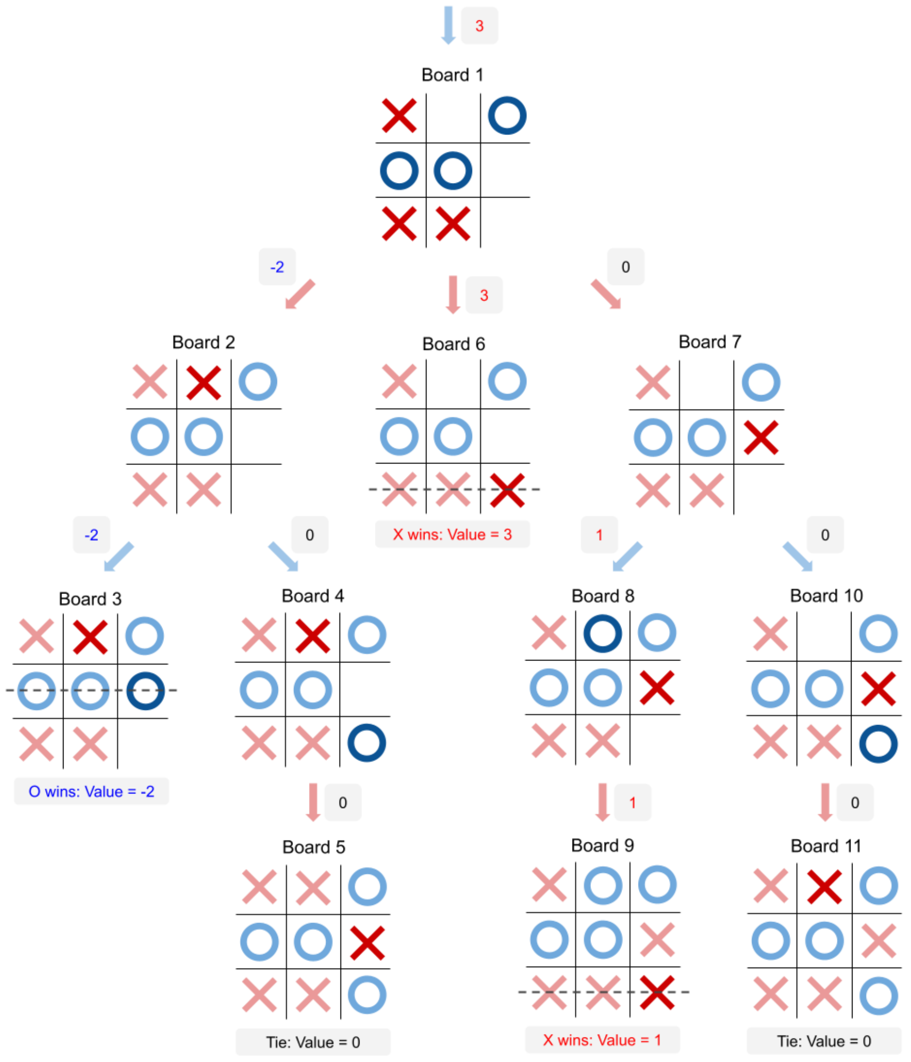

Figure 2: The tic-tac-toe tree below the shown starting configuration (labeled Board 1). This tree shows all possible games that stem from the configuration achieved in Board 1. Red arrows represent possible moves by X, and blue arrows represent possible moves by O. The numbers in grey boxes represent the weight given to that choice by the Minimax algorithm.

2.3 Possibilities for Implementation

We were originally going to implement the algorithm described in section 2.2 (and actually started writing the Matlab code for it), but realized a full implementation and write-up for something we just learned about might be over-scoped for the time we had to complete the assignment. Instead, we will explain how we planned to implement the algorithm.

Tic-Tac-Toe has a relatively small amount of possible board states so we were going to initially generate a tree that mapped out all the possible game states and the paths to get to each. Once we had this, the plan was to write a recursive function to go through and assign values to each node using the Minimax algorithm as explained above. From here we would be able to use that tree to create a bot that we could play Tic-Tac-Toe against and, if we did everything correctly, we should not be able to ever win against it.

2.3.1 Optimizations

Minimax by itself, while very accurate, is not the most efficient algorithm. There are certain optimizations that you can make that would make the algorithm run much faster. Below are two such optimizations that we looked into.

Alpha-Beta Pruning: Alpha-Beta pruning is a way to selectively cut branches (and their respective sub trees) off of the tree which can lead to a significant speed up in the computation time. To implement this optimization, you need to start keeping track of the maximum and minimum available options at each node of the tree. If, while you are calculating the value of a node, you run in to a case where there can't be a better value in the rest of the subtree, you can prune that subtree off the parent tree and not devote any computation time to it.

Memoization: Memoization can be used to speed up the recursive element of the Minimax algorithm by caching the outcome of running minimax on a subtree. If we ever run into that subtree again anywhere in the tree, and we will since there are multiple ways to arrive at the same board state, we already have the values for the subtree stored so instead of re-calculating the whole thing, we just have to fill it in. This can be further used to speed up the computation more when we factor in board rotations. If we take any board state and rotate it 90, 180, or 270 degrees, the board looks completely different to the algorithm but the subtrees would essentially be isomorphic (the same values would be found in the subtree just in different orders). Using memoization, we could also take these cases into account and speed the computation up even more.

3 Resources

Huffman Coding Wikipedia
Canonical Huffman Coding Wikipedia
Scranton Canonical Huffman Coding Page

Minimax Wikipedia
Kylie Ying's Video
Sebastian Lague's Video
Geeks for Geeks Alpha-Beta Pruning

4 Appendix

4.1 Program for encoding and decoding byte files using a canonical Huffman code

```python
# huffman.py

from heapdict import heapdict
from collections import Counter

CHAR_BITS = 8
MAX_CODE_LEN = CHAR_BITS * 2
MAX_CHARS = 2 ** CHAR_BITS


class Node:
    def __init__(self, char, children = []):
        self.alphabet = ["0", "1"]
        if len(children) > len(self.alphabet):
            raise
        self.char = char
        self.children = {code_char: child for code_char, child in zip(self.
    alphabet, children)}

    def __repr__(self):
        return str(self.char)

    def __hash__(self):
        return hash(self.char)

    def print(self, code_word=""):
        for code_char, child in self.children.items():
            child.print("  " + code_word + str(code_char))


def huffman_encode(original):
    root = make_huffman_tree(original)
    code_dict = {}
    make_code(root, "", code_dict)
    code_dict = canonize_encoding(code_dict)
    encoding_string = gen_canonical_code_string(code_dict)
    encoded = encode_orig(original, code_dict)
    return encoding_string + encoded


def huffman_decode(encoded):
    encoded_msg, min_cw, cw_to_char = reconstruct_table_from_encoded(encoded)
    return reconstruct_orig_from_table(encoded_msg, min_cw, cw_to_char)


def make_huffman_tree(original):
    alphabet=["0", "1"]
    freqs = Counter(list(original))
    priority_queue = heapdict()
    for char, freq in freqs.items():
        node = Node(char)
        priority_queue[node] = freq
    while len(priority_queue) > 1:
        freq_sum = 0
        children = []
        for code_char in alphabet:
            if len(priority_queue) == 0:
                break
            child_node, freq = priority_queue.popitem()
            freq_sum += freq
            children.append(child_node)
        parent_node = Node(None, children)
        priority_queue[parent_node] = freq_sum
    return list(priority_queue)[0]


def make_code(node, code_word="", dictionary=None):
    if node.char:
        dictionary[node.char] = code_word
    else:
        for code_char, child in node.children.items():
            make_code(child, code_word + str(code_char), dictionary)


def canonize_encoding(code_dict):
    dictionary = {}
    pairs = list(code_dict.items())
    pairs.sort(key=lambda item:item[0])
    pairs.sort(key=lambda item:len(item[1]))
    curr_code_word_decimal = 0
    last_len = 0
    for char, code_word in pairs:
        orig_len = len(code_word)
        new_code_word = bin(curr_code_word_decimal)[2:]
        if orig_len > last_len:
            new_code_word += "0" * (orig_len - last_len)
        new_code_word = new_code_word.zfill(orig_len)
        dictionary[char] = new_code_word
        last_len = orig_len
        curr_code_word_decimal = int(new_code_word, 2)
        curr_code_word_decimal += 1
    return dictionary

def gen_canonical_code_string(code_dict):
    cw_to_char_str = ""
    counter = 0
    pairs = list(code_dict.items())
    pairs.sort(key=lambda item: item[0])
    min_cw = ["1" * i for i in range(1, MAX_CODE_LEN + 1)]
    while counter < MAX_CHARS and len(pairs) > 0:
        if pairs[0][0] == chr(counter):
            char, code_word = pairs.pop(0)
            length = len(code_word)
            min_cw[length-1] = min(code_word, min_cw[length-1])
            cw_to_char_str += f"{length:04b}"
        else:
            cw_to_char_str += "0" * 4
        counter += 1
    min_cw_str = "".join(min_cw)
    cw_to_char_str += "0" * (MAX_CHARS * 4 - len(cw_to_char_str))
    return min_cw_str + cw_to_char_str


def encode_orig(original, code_dict):
    encoded = ""
    for char in original:
        encoded += code_dict[char]
    return encoded


def reconstruct_table_from_encoded(encoded):
    min_cw_str_len = MAX_CODE_LEN * (MAX_CODE_LEN + 1) // 2
    min_cw_str = encoded[:min_cw_str_len]
    cw_to_char_str = encoded[min_cw_str_len:min_cw_str_len+4*MAX_CHARS]
    encoded_msg = encoded[min_cw_str_len+4*MAX_CHARS:]
    min_cw = []
    for cw_len in range(1, MAX_CODE_LEN+1):
        code_word, min_cw_str = min_cw_str[:cw_len], min_cw_str[cw_len:]
        min_cw.append(code_word)
    cw_to_char = [[] for i in range(MAX_CODE_LEN)]
    for i in range(MAX_CHARS): # for every character in the key
        char = chr(i)
        bitlength_chunk = cw_to_char_str[i*4:i*4+4]
        bitlength = int(bitlength_chunk, 2)
        if bitlength > 0:
            cw_to_char[bitlength-1].append(char)
    return encoded_msg, min_cw, cw_to_char


def reconstruct_orig_from_table(encoded_msg, min_cw, cw_to_char):
    idx = 0
    cw_buffer = ""
    original = ""
    while idx < len(encoded_msg):
        cw_buffer += encoded_msg[idx]
        idx += 1
        # if value of code_word is at least value of min code_word
        length = len(cw_buffer)
        i = length - 1
        j = int(cw_buffer, 2) - int(min_cw[i], 2)
        if 0 <= j < len(cw_to_char[i]):
            char = cw_to_char[i][j]
            original += char
            cw_buffer = ""
    return original


def bitstring_to_bytes(s):
    # From https://stackoverflow.com/questions/32675679/convert-binary-string-to-bytearray-in-python-3
    return int(s, 2).to_bytes((len(s) + 7) // 8, byteorder="big")


fname = input("Enter a file name: ")
with open(fname, "r+") as f:
    original = f.read()
    encoded = huffman_encode(original)
    recovered = huffman_decode(encoded)

name, ext = fname.split(".")

fname_encoded = name + "_encoded" + "." + ext
with open(fname_encoded, "wb+") as f:
    f.write(bitstring_to_bytes(encoded))

fname_recovered = name + "_recovered" + "." + ext
with open(fname_recovered, "w+") as f:
    f.write(recovered)
```

4.2 Recursive function for the creation of the complete tic-tac-toe game tree

We wrote this Matlab function before we realized that implementing the entire Minimax algorithm for tic-tac-toe would be over-scoped for this project.

```matlab
function addChildBoards(parent,parentBoardIndex)
    % parent: 3x3 uint8 matrix representing a board (0=blank, 1=X, 2=O)
    % parentBoardIndex: int representing the location of "parent" in "allBoards"
    % addChildBoards() is a recursive function that takes in a parent board
    % and its index, and adds all of its child boards to the master list of
    % boards "allBoards" and records the indexes of the parent board for each
    % board it creates in the "parentIndexes" array

    global nextBlankBoard allBoards parentIndexes

    % Find the number of 0's in the parent board
    numBlanks = length(find(~parent));

    % If there are no open spaces, end the game
    if numBlanks < 1
        return
    end

    % Choose who's turn it is (1==X 2==O)
    if (mod(numBlanks,2) == 1)
        newChar = 1;
    else
        newChar = 2;
    end

    % Itterate through board
    for i = 1:3
        for j = 1:3
            % Check if spot is empty
            if (parent(i,j) == 0)
                % Clone parent board as a new node
                newLayer = uint8(parent);
                % Add next move to selected spot
                newLayer(i,j) = newChar;
                % Add new board to allBoards list
                allBoards(:,:,nextBlankBoard) = newLayer;
                % Record the parent board of the new board
                parentIndexes(nextBlankBoard) = parentBoardIndex;
                nextBlankBoard = nextBlankBoard + 1;

                % Run function again on the child
                addChildBoards(newLayer,nextBlankBoard-1);
            end
        end
    end
end
```