Deterministic Models and Optimization:

Programming projects

Reid Falconer[a], Ari Lam[a], Evelyn Maria Molina Bolanos[a]

[a]*Barcelona Graduate School of Economics, Barcelona, Spain*

In this report, we analyse two algorithms, namely the Edit distance to calculate the least number of edit operations that are necessary to modify one string into another and Huffman codes to construct an optimal prefix code given input text. For each of these algorithms, we provide a brief overview of their key concepts, present their pseudo code, outline a proof of correctness, and conduct a complexity analysis. Furthermore, the Python code is available for both algorithms.

1. Edit Distance

1.1. Algorithm Overview

In computer science, edit distance is a way of quantifying how dissimilar two strings are to one another by calculating the minimum number of edit operations necessary to transform one string into the other. One of the simplest sets of edit operations is that defined by Levenshtein (1966) which consist of three operations on a string: (i) I: Insertion of a character, (ii) D: Deletion of a character, and (iii) S: Substitution of a character.

Each operation has an imputed cost[1], and subsequently the algorithm strives to minimise the cost of edit operations (i.e.the number of operations $\{D, I, S\}$).

Email addresses: reid.falconer@bracelonagse.eu (Reid Falconer), ari.lam@barcelonagse.eu (Ari Lam), evelyn.molinabolanos@barcelonagse.eu (Evelyn Maria Molina Bolanos)

[1]In Levenshtein's definition, each of these operations has unit cost (except substitution of a single symbol by itself has zero cost).

For example if each operation has a unit cost and we started with the sequence

$$X = \langle A, B, C, A, D, A \rangle$$

we could convert it to

$$Y = \langle A, B, A, D, C \rangle$$

with three edit operations (i.e. delete the C, delete the last A and insert a C). This is the best that can be done, so the minimum edit distance is 3. Nevertheless, the notion of edit distance can be generalised to allow different weights for alternative edit operations.

To compute this algorithm, a dynamic programming approach is used. Specifically, this technique allows one to construct the solution of a larger problem instance by composing solutions to smaller instances, and the solution to each smaller instance can be used in multiple larger instances(Agrawal, 2017).

Furthermore, edit distances find applications in many different fields, for example in bioinformatics; the edit distance can be used to quantify the similarity of DNA and protein sequences.

1.2. Pseudo code

To program this algorithm the characters of the strings s_1 and s_2 are displayed in an array form. The algorithm fills the entries in a matrix whose two dimensions equal the lengths of the two strings(Standford, n.d.). Each jump horizontally or vertically, in the matrix, corresponds to an insertion or a deletion, respectively. Since the algorithm takes into consideration the cost of each operation the final output always minimizes the cost locally.

```
m = len(str1)
n = len(str2)
# Creating an array, storing the results from the subproblem
D = [[0 for x in range(n+1)] for x in range(m+1)]
  # Filling the list from top to bottom
  for i in range(m+1):
```

```
for j in range(n+1):
    if i == 0:
        D[i][j] = j  # insertion
    elif j == 0:
        D[i][j] = i    #deletion
    elif str1[i-1] == str2[j-1]:
        D[i][j] = D[i-1][j-1]
    else:
        D[i][j] = min(D[i][j-1] + 1, # insertion
            D[i-1][j] + 1,    # deletion
            D[i-1][j-1]+1) # substitution
```

1.3. Proof of Correctness

Since the algorithm is built recursively, the standard technique for proving its correctness is induction. Our claim is that any alignment of strings s_1, s_2 satisfying the recursion relation is done at a minimal cost. Following Aicher and Hand (2013) the proof of correctness considers the following:

- Base case: The edit distance between any string, s (i.e. either string X or Y) and an empty string is $len(s)$, i.e. just insert all the characters in s or drop all characters in s.

- Inductive cases: If we assume we have calculated the minimal alignment for $OPT(k, l)$ for $k < i, l < j$, where $OPT(i, j)$ is the minimum cost of aligning the substrings $s_1[1 \dots i]$ and $s_2[1 \dots j]$, then there are only 3 possible previous sub-alignments based on the last operator(s):

    1. Substitution: We substitute $s_1[i]$ for $s_2[j]$, with a cost equal to $c(Sub)$. The rest of the alignment cost is from aligning $s_1[1 \dots i-1]$ and $s_2[1 \dots j-1]$. Therefore the minimum cost ending with a substitution is $OPT(i-1, j-1) + c(Sub)$.

    2. Delete and insert: We add a gap character after $s_2[j]$ to match $s_1[i]$ with a cost of $c(InDel)$. The rest of the alignment cost is from aligning $s_1[1 \dots i-1]$ and

$s_2[1 \ldots j]$. Therefore the minimum cost ending with a substitution is $OPT(i-1, j) + c(Sub)$

3. Insert and delete: Similarly, we add a gap character after $s_1[i]$ to match $s_1[j]$ with a cost of $c(InDel)$. The rest of the alignment cost is from aligning $s_1[1 \ldots i]$ and $s_2[1 \ldots j-1]$. Therefore the minimum cost ending with a substitution is $OPT(i, j1) + c(Sub)$

- Taking the minimum over the possible operations gives the recursive relation. Since these are the only possible paths to aligning substrings (i, j), the recursion gives the minimal cost for aligning (i, j).

1.4. Complexity

Given that s_1 is always a suffix of the first input string and that s_2 is always a suffix of the second input string. Since there are $n+1$ suffixes of the first input string and $m+1$ suffixes of the second input strings, the total number of sub-problems is $(n+1)(m+1)$. Therefore the time-complexity of the algorithm is $\mathcal{O}(nm)$ where m and n are the lengths of the two strings (s_1 and s_2) respectively[2].

1.5. Task Results

1. DNA

   X = ACTACTAGATTACTTACGGATCAGGTACTTTAGAGGCTTGCAACCA

   Y = TACTAGCTTACTTACCCATCAGGTTTTAGAGATGGCAACCA

2. Proteins

   X = AASRPRSGVPAQSDSDPCQNLAATPIPSRPPSSQSCQKCRADARQGRWGP

   Y = SGAPGQRGEPGPQGHAGAPGPPGPPGSDG

[2]It takes $\mathcal{O}(nm)$ time to fill in the $n * m$ dynamic programming matrix: the pseudocode consists of a nested 'for' loop inside of another 'for' loop.

Table 1: Edit Distance and penalized Edit Distance for DNA and protein sequences X and Y

	DNA	**Proteins**
Edit Distance (X, Y)	10	37
Penalty Edit Distance (X,Y)	15	58

2. Huffman codes

2.1. Algorithm Overview

Huffman coding is a lossless data compression algorithm. It assigns variable-length codes to each symbol, lengths of the assigned codes are based on the frequencies of corresponding symbols. The smallest code is assigned to the most frequent symbol and vice versa (Kleinberg and Tardos, 2006).

Prefix Codes, which the bit sequences are assigned in such a way that the code assigned to one symbol is different from any other symbols, are assigned so that Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

To compute this algorithm, a greedy approach is used. This technique looks at the frequency of each character at each round and stores it as a binary string in an optimal way (i.e. at each step, the algorithm makes a "greedy" decision to merge the two subtrees with least weight).

2.2. Pseudo code

The pseudocode for Huffmans algorithm is given below. Let C denote the set of characters, and let $n = |C|$. Each character $x \in C$ is associated with an occurrence probability $prob[x]$. Initially, the characters are all stored in a priority queue Q. Recall that this data structure can be built initially in $\mathcal{O}(n)$ time, and we can extract the element with the smallest key in $\mathcal{O}(n \log n)$ time and insert a new element in $\mathcal{O}(n \log n)$ time. The objects in Q are sorted by probability. Note that with each execution of the for-loop, the number of items in the queue decreases by one. So, after $n - 1$ iterations, there is exactly one

element left in the queue, and this is the root of the final prefix code tree (Mount, 2017).

```
huffman(C, prob) { # C = chars, prob = probabilities
    for each (x in C) {
        add x to Q sorted by prob[x] # add all to priority queue
    }
    for (i = 1 to |C| − 1) { # repeat until only 1 item in queue
        z = new internal tree node
        left[z] = x = extract−min from Q # extract min probabilities
        right[z] = y = extract−min from Q
        prob[z] = prob[x] + prob[y] # z's probability is their sum
        insert z into Q # z replaces x and y
    }
    return the last element left in Q as the root
}
```

2.3. Proof of Correctness

We prove the correctness of Huffmans algorithm by following Hadzilacos (2017) who use induction on the number of symbols n in the alphabet. The base case, $n = 2$ is evident because the only possibility (that is not suboptimal) is a code where both codewords are one bit long, which is what Huffmans algorithm produces in this case. Suppose that the algorithm produces an optimal tree for alphabets with $n - 1 \geq 2$ symbols and their associated frequencies. We will prove that it provides an optimal tree for alphabets with n symbols and their associated frequencies.

Let A be an alphabet with n symbols, and $f(a)$ be the frequency for each $a \in A$. Let H be the tree produced by Huffmans algorithm for A, f. We must prove that H is optimal for this input. By the algorithm, there are two symbols of minimum frequency (according to f) that are siblings in H; let these symbols be x and y. Let z be a new symbol (that is not in A); and let $A' = (A - \{x, y\}) \cup z$ and f' be frequencies of the symbols in A' defined

by :

$$f'(a) = \begin{cases} f(a), & \text{if } a \neq z \\ f(x) + f(y), & \text{if } a = z \end{cases}$$

Finally, let H' be the tree obtained from H by removing x and y and replacing their parent by z. From the definition of weighted average depth (ad), we have

$$\mathbf{ad}(H) = \mathbf{ad}\,(H') + (f(x) + f(y))$$

Note that H' is a tree produced by Huffman's algorithm on input A', f'. A' has $n-1$ symbols so, by induction hypothesis

$$H' \text{ is optimal for } A', f'$$

Now, let T be an optimal tree for A, f. Without loss of generality, we can assume that x and y are siblings and are at maximum depth of T. Let T' be obtained from T as H' was obtained from H. Thus, T' is a tree for A' , f' . We have by definition of **ad**:

$$\begin{aligned} \mathbf{ad}(T) &= \mathbf{ad}\,(T') + (f(x) + f(y)) \\ &\geq \mathbf{ad}\,(H') + (f(x) + f(y)) \\ &= \mathbf{ad}(H) \end{aligned}$$

Since T is optimal for A, f, so is H. So, Huffman's algorithm produces optimal trees for alphabets with n symbols and their associated frequencies.

Furthermore, two claims are proved intuitively below:

1. In an optimum code, symbols that occur more frequently will have shorter codewords than symbols that occur less frequently.

Proof: Given that more frequent symbols have smaller codes and vice versa, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum. Therefore, this first statement is true.

2. In an optimum code, the two symbols that occur least frequently will have the same length.

Proof: Lets make an assumption that an optimum code exists for the two least frequent symbols but Code2 has a longer than Code1 (e.g. Code1: 01001, Code2: 0100001). Due to the property of prefix code, even if we drop the last 2 bits of Code2, Code 1 and Code 2 would still be distinct (i.e. Code1: 01001, Code2: 01000). Since they are the least frequent symbols, nothing can be longer than them which means the shortened Code2 would never become the prefix of other codewords. Now we have a shorter length than the previous one but this violates our initial assumption. Therefore, this second statement is also true.

2.4. Complexity

The complexity is $\mathcal{O}(n \log n)$. There are $\mathcal{O}(n)$ iterations for each symbol. Therefore, using a heap to store the trees, each iteration requires $\mathcal{O}(n \log n)$ time to determine the lowest frequent symbols and insert that tree.

2.5. Task Results

2.5.1. Compression Rates

The code obtained from the Huffman algorithm is a variable-length code since the code-lengths vary from letter to letter. When there is a constant l_0 such that $l(c) = l_0$ for all $c \in C$, it is called a fixed-length code. To calculate the compression rate obtained from the Huffman code, we compared the code-length of a given input text in a fixed-length code with the code-length of Huffman's variable-length code. The minimum number of bits necessary to code both the alphabets is $I_0 = 8$ (i.e. UTF-8). So the compression rate of encoding the text is given by:

$$Comp = \left(1 - \frac{Codelength}{l_0 \times Textlength}\right) \times 100$$

where $Codelength$ is the length of the text in bits when coded using Huffman encoding and $Textlength$ is the number of letters and symbols in the text.

When evaluating the compression rates across the two texts (see Table 2) we find that both texts have almost identical compressibility rates with the Hamlet text and Goethe text having compression rates of 46.87% and 46.89% respectively.

Table 2: Compression rates across texts using the standard distribution for the two sample texts. *Textlength* is measured in number of letters and *Codelength*, in bits

	Hamlet	**Goethe**
Text Length	802	1156
Code Length	3402	4903
Compression Rate (*Comp*)	46.84%	46.89%

Table 3: Huffman Text Encoding

Hamlet		**Goethe**	
Huffman Code	Alphabet	Huffman Code	Alphabet
00	␣	101	e
0101	i	111	␣
0110	t	0000	u
1000	l	0001	a
1001	o	0011	s
1011	a	0111	r
1110	e	1000	h
01001	y	1010	i
01110	,	1011	n
01111	d	01100	l
11000	h	10010	m
11001	r	10011	t
11011	m	11001	c
11110	n	11011	d
11111	s	001000	z
010000	v	001001	f
010001	f	001010	g
101001	w	011010	k
101010	u	011011	b
110100	b	110000	o
1010001	.	110001	w
1010110	c	110101	,
1101010	p	00101101	!
1101011	!	00101110	.
10100000	’	00101111	ö
10100001	k	11010000	;
10101111	g	11010001	ü
101011100	x	11010010	p
101011101	?	001011000	q
		001011001	ä
		110100110	j
		110100111	v

3. References

Agrawal, K. (2017), 'Dynamic programming'.

URL: *http://www.classes.cec.wustl.edu/ cse341/web/handouts/lecture11.pdf*

Aicher, C. and Hand, R. (2013), 'Sequence alignment lecture'.

URL: *http://tuvalu.santafe.edu/ aaronc/courses/5454/csci5454*$_s pring 2013_C SL3.pdf$

Hadzilacos, V. (2017), 'Correctness and running time of huffman's algorithm'.

URL: *http://www.cs.toronto.edu/ vassos/teaching/c73/handouts/huffman.pdf*

Kleinberg, J. and Tardos, E. (2006), *Algorithm design*, Pearson Education India.

Levenshtein, V. I. (1966), Binary codes capable of correcting deletions, insertions, and reversals, *in* 'Soviet physics doklady', Vol. 10, pp. 707–710.

Mount, D. (2017), 'Greedy algorithms: Huffman coding'.

URL: *http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect06-greedy-huffman.pdf*

Standford (n.d.), 'Edit distance'.

URL: *https://nlp.stanford.edu/IR-book/html/htmledition/edit-distance-1.html*