On prefix codes satisfying RLL-constraint for Zipf distribution

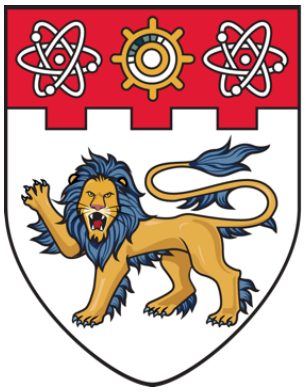
Ho, Shaun

2020

Ho, S. (2020). On prefix codes satisfying RLL-constraint for Zipf distribution. Master's thesis, Nanyang Technological University, Singapore.

https://hdl.handle.net/10356/143338

https://doi.org/10.32657/10356/143338



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

ON PREFIX CODES SATISFYING RLL-CONSTRAINT FOR ZIPF DISTRIBUTION

SHAUN HO

SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES

2020

ON PREFIX CODES SATISFYING RLL-CONSTRAINT FOR ZIPF DISTRIBUTION

SHAUN HO

SCHOOL OF PHYSICAL AND MATHEMATICAL SCIENCES

A thesis submitted to the Nanyang Technological University in partial fulfillment of the requirements for the degree of Master of Science

2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research done by me except where otherwise stated in this thesis. The thesis work has not been submitted for a degree or professional qualification to any other university or institution. I declare that this thesis is written by myself and is free of plagiarism and of sufficient grammatical clarity to be examined. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

26/3/2020

.....

Date

.....

Shaun Ho

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it of sufficient grammatical clarity to be examined. To the best of my knowledge, the thesis is free of plagiarism and the research and writing are those of the candidate's except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

26/3/2020

.....

Date

.....

Kiah Han Mao

Authorship Attribution Statement

This thesis does not contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

26/3/2020

.....

Date

.....

Shaun Ho

Acknowledgements

I wish to express my greatest gratitude to my supervisor, Assistant Professor Kiah Han Mao of School of Physical and Mathematical Sciences at Nanyang Technological University, for giving me the opportunity to work with him on the field of Information Theory. His apt guidance and exceptional advice laid the groundwork in order for us to attain the results of this thesis, and has provided me with a fulfilling and enriching Masters programme.

– Shaun Ho

Abstract

The study of Information Theory has been ubiquitously applied to many fields in the digital world, with significant focus in transmitting data through channels as efficiently as possible. Therein lies the Optimal Coding Problem; where symbols of varying probabilities of occuring in the source text are encoded to as few bits as possible. In this thesis, we will focus on special type of codes which are bound by the Runlength-limited constraint. We apply this constraint to familiar optimal compression algorithms, like Huffman coding, as well as construct our own version of these constraint codes, with help from a dynamic programming algorithm made by Golin. The results obtained from testing the efficiency of these codes showed that Golin's construction is more efficient than Huffman coding with the Runlength-limited constraint, in terms of optimality. Finally, we further analyse the structures of coding trees that follow the Runlength-limited constraint and show that such trees produce very particularly unique patterns.

Contents

Symbols and Acronyms

$\mathcal{Q}$	the set of symbols in the source text
Σ	the encoded alphabet
W	the set of codewords or code
C	the average number of bits per symbol average cost
RLL	abbreviation for runlength limited
$F_k(n)$	the set of all k-RLL codewords of length n
$(u; l_1, l_2, ..., l_k)$	the tree signature of a tree at level i
C_H	average cost of a k-RLL constrained Huffman code
C_G	average cost of a k-RLL code constructed by Golin's algorithm
Ψ_k	the unique positive root such that $\Psi_k^k = 1 - \Psi_k - \Psi_k^2 - ... - \Psi_k^{k-1}$

Chapter 1

Introduction

1.1 Background and Motivation

In 1948, the publication of Shannon's prominent paper 'A Mathematical Theory of Communication' led to a paradigm shift of how mankind communicated with one another. Information theory gradually became an ubiquitous metric to study the propagation, storage and transmission of information.

Notably, its application in computer science and fields of engineering shed light on the processes of data compression and source coding. From the compact disc, to sending signals into deep space during the Voyager missions, information theory is undoubtedly essential to determine the unending potential of data transmission, and sometimes its limits.

Later years saw the development of lossless data compression, which allows the source text to be reconstructed perfectly after being sent through a channel. In Chapter 2, we revisit the so-called Huffman coding, a staple method of lossless data compression algorithm based on the frequency of symbols in the source text. Huffman coding is known to be optimal for symbol-by-symbol encoding, thus we will add an additional constraint of only permitting so-called Runlength-limited codes.

We also construct a code based off of Golin's paper 'A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes with Unequal Letter Costs'

[1]. These codes also admit the same constraint, and are shown to be more efficient than Huffman codes with those constraints.

In Chapter 3, we further analyse the construction of these codes, and predict its codeword structure with varying probability distributions of the frequency of symbols in the source text.

1.2 Preliminaries

In data compression and source coding, symbols in the source text follow a discrete distribution of probabilities, i.e. the frequency of how much they appear in the source text. When transmitting the source text over a channel, we are concerned with sending as few bits per symbol as possible. It is thus common practice to encode the more probable symbols to codewords with fewer bits and less probable symbols to codewords with more bits.

Formally, let $\mathcal{Q} = \{a_1, a_2, ..., a_q\}$ be the set of source symbols of size q from the source text. Let $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_r\}$ be the *encoding alphabet* consisting of r many *characters*. An arbitrary concatenation of said characters in Σ is called a codeword w. The set of codewords $w \in W$ is called a *code*. We map or *encode* each symbol from the source text to our code.

Example 1.1. $\mathcal{Q} = \{t, e, x\}$, $\Sigma = \{0, 1\}$, $W = \{0, 1, 01\}$

Then the encoding

$$t \to 0$$

$$e \to 1$$

$$x \to 01$$

of the source text *text* returns the string 01010.

Let $c_1, c_2, ..., c_r$ be integers such that $c_i = length(\alpha_i)$ is the bit length of each character in Σ, or *cost*. Since each codeword is just a concatenation of the encoding

alphabet, we extend the definition of cost of each codeword $c(w_1), c(w_2), ..., c(w_q)$ with

$$c(w_i) = \sum_{j=1}^{k} c_{i_j}, \text{ where } w_i = \alpha_{i_1}\alpha_{i_2}...\alpha_{i_k}$$

Now each symbol in the source text is assigned real valued probabilities $p_1, p_2, ..., p_q$ such that

$$\sum_{i=1}^{q} p_i = 1$$

For the remainder of this thesis, we assume $p_1 \geq p_2 \geq ... \geq p_q > 0$ and $0 < c_1 \leq c_2 \leq ... \leq c_r$. This means that for $\mathcal{Q}$ we fix the probability of source symbols from most frequent to least frequent and for Σ, the encoding alphabet from shortest to longest length.

Finally, let $C(W)$ be defined as the *average cost* of a code, where

$$C(W) = \sum_{i=1}^{q} p_i c(w_i)$$

This is also known as the average number of bits per symbol. We will use average cost for consistency and it will be more relevant in Chapter 3. We also have the following definition for average cost of a code.

Definition 1.2.1 (Optimality)**.** Given a fixed $\mathcal{Q}$, $p_1, p_2, ..., p_q$, and $c_1, c_2, ..., c_r$, a code W_1 is more *efficient* than W_2 if $C(W_1) < C(W_2)$. A code W is *optimal* if it is the most efficient i.e. $C(W) \leq C(W_i)$ for all i.

The goal of data compression is to find codes that are as efficient as possible, i.e. codes with the least possible average cost. This is known as the Optimal Coding Problem.

In Example 1.1, the encoding does not admit a bijection between the source text and the encoded string: It is also possible to decode the string 01010 into *tetet*. This could be modestly solved by adding a distinct marker between each codeword

i.e. $0\big|1\big|01\big|0$. However, these distinct markers would cost extra bits, and becomes less efficient as the source text grows larger. For this, we shall introduce the so-called prefix codes.

Definition 1.2.2 (Prefix Code)**.** A *prefix code* or prefix-free code is a type of code where no codeword is a prefix of another codeword.

Example 1.2. The following codes

$$\{101, 110, 011\}$$

$$\{1, 01, 001, 0001\}$$

are prefix codes but

$$\{0, 1, 01, 010\}$$

is not a prefix code as 0 is a prefix of 01.

Prefix codes enjoy a property of being *uniquely decodable*, that is, we can retrieve the source text from a continuous string of codewords without the need for any special markers between each codeword. In Chapter 2, we see that Huffman coding is an example of prefix coding.

We will now define a special case of codes consistent with Manickam and Kashyap's paper [2]. These codes will serve as the additional constraint of the Optimal Coding Problem in Chapter 2 and Chapter 3.

Definition 1.2.3 (Runlength-limited code and the Runlength-limited constraint)**.** A binary codeword is *runlength-limited* (RLL) if each succeeding character in the encoding alphabet is an increasing string (or *run*) of zeroes followed by a one, i.e. $\Sigma = \{1, 01, 001, 0001, ...\}$.

A codeword satisfies the *k-RLL constraint* if its encoding alphabet has exactly $k+1$ characters, with the longest character having k zeroes followed by a one, i.e. $\Sigma = \{1, 01, 001, ..., 0^k1\}$.

A code is called a *k-RLL code* if all its codewords satisfy the k-RLL constraint.

Example 1.3.

$$W = \{1, 01, 0101, 001, 001001\}$$

is a 2-RLL code.

This method of encoding is commonly used in hard disk drives, and may also be written as a RLL(0,k) code. Since this definition of k-RLL codes uses only binary letters, for the remainder of this thesis, we will restrict our scope to binary codewords.

With careful construction, we can arrive at the following lemma:

Lemma 1.2.1. There exists a k-RLL code that is also a prefix code.

Proof. Let T be a full $k+1$-ary tree with every parent node having either 0 or $k+1$ children (excluding the root which always has $k+1$ children). Starting with the root, label each branch of the root with exactly one character in Σ. Repeat this process for each internal node until all branches are labelled. Then, construct a codeword by travelling from the root to each leaf, concatenating each character as we traverse down the tree, creating a codeword that satisfies the k-RLL constraint. Since a path from the root to each leaf in T is unique, the codeword cannot be a prefix of another. The set of all paths from the root to every leaf in T is thus a prefix k-RLL code. □

In Chapter 3, we will be using this code construction by creating a full tree with q leaves, each leaf i being assigned a probability p_i. Each branch is then assigned an encoding alphabet α_i with cost c_i. Now, if we traverse from the root to a leaf i, summing all costs along its path, we get the cost of the codeword $c(w_i)$. We then take the product of the probability p_i and repeat this for each distinct path from the root to every leaf in order to obtain the average cost of the code $C(W)$. Note that if the tree is not full, we get a tree that is visually similar to Huffman coding.

Example 1.4. In Figure 1.1., we have two different full trees representing two different codewords. Given $p_1 = p_2 = 0.4$, $p_3 = 0.2$ and $c_1 = 1$, $c_2 = 2$ We determine the average cost of tree (a) to be $0.4 + 3(0.4) + 4(0.2) = 2.4$ and tree (b) to be $2(0.4) + 2(0.4) + 3(0.2) = 2.2$. Thus, (b) is more efficient.

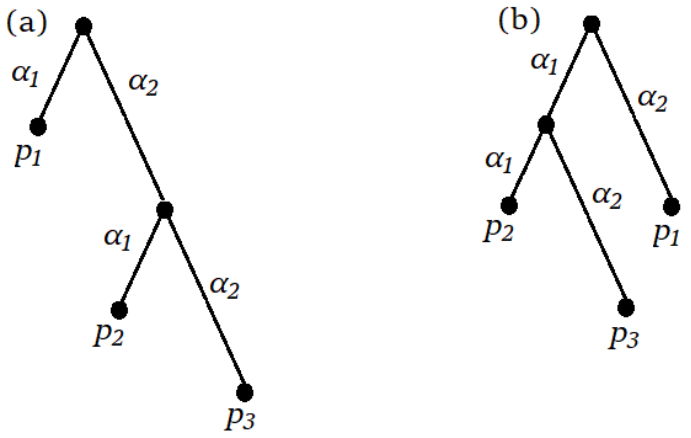


FIGURE 1.1: Two trees with representing two different mappings of codewords. Tree (a) encodes the source text to the code $\{\alpha_1, \alpha_2\alpha_1, \alpha_2\alpha_2\}$, while tree (b) has the code $\{\alpha_1\alpha_1, \alpha_1\alpha_2, \alpha_2\}$. For ease of visualisation, the branch α_2 has twice the depth of α_1, mirroring their costs.

It is worthy to point out that if we used a different probability distribution, say $p_1 = 0.5$, $p_2 = 0.4$, $p_3 = 0.1$, tree (a) would be more efficient instead. We explore this in greater detail in Chapter 3.

Chapter 2

Huffman coding with k-RLL constraint

In this chapter we will briefly describe how Huffman coding works and apply the k-RLL constraint. After which, we will show some results between Huffman coding and Golin's construction method.

Huffman coding is a type variable-length encoding algorithm that generates a prefix code based on the probability of each symbol in the source text. The so-called greedy algorithm constructs a tree from the bottom-up by picking the pair of symbols having the lowest pair of probabilities at each step of construction. This is repeated until all pairs of probabilities have been used up and the resulting tree is a binary tree, similar to the tree described in Lemma 1.2.1.

We describe the algorithm formally as follows:

1. Let $p_1, p_2, ..., p_q$ be a forest of q single node trees, such that each leaf is assigned a real valued probability $0 < p_i < 1$.
2. While there is at least one node in the forest:
   (a) Remove the two nodes of lowest probabilities from the forest.
   (b) Create a new node that will be the parent of the two removed nodes, such that this node is the root, with its value being the sum of the probabilities of its children. Label the branch of the left child '0' and the branch of the right child '1'.

(c) Add this root node and its two children (if the children are not added yet) to the Huffman code tree. Add this root node (only the root node and not its children) to the forest as well. Repeat step 2.

3. Return the Huffman code tree.

Example 2.1. Let $\mathcal{Q} = \{a_1, a_2, a_3, a_4, a_5\}$ be symbols from a source text and we are given the probabilities $p_1 = 0.31$, $p_2 = 0.26$, $p_3 = 0.16$, $p_4 = 0.15$, $p_5 = 0.12$, where p_i is the respective probability for a_i. Figure 2.1 shows how the tree is constructed accordingly. We will get the resulting encoding

$$a_1 \to 00$$
$$a_2 \to 10$$
$$a_3 \to 11$$
$$a_4 \to 010$$
$$a_5 \to 011$$

It is clear that Huffman coding generates a prefix code, but not a k-RLL code, as it is a tree identical to our previous lemma.

Huffman coding is also known to be optimal for symbol-by-symbol encoding. We will state this optimality without proof.

Lemma 2.0.1 (Optimality of Huffman Coding)**.** Let $p_1, p_2, ..., p_q$ be a probability distribution over q source symbols. Let H_q be the code that encodes these source symbols via the Huffman coding algorithm. Then H_q is optimal.

We will now show how to apply the k-RLL constraint on codes generated by this algorithm.

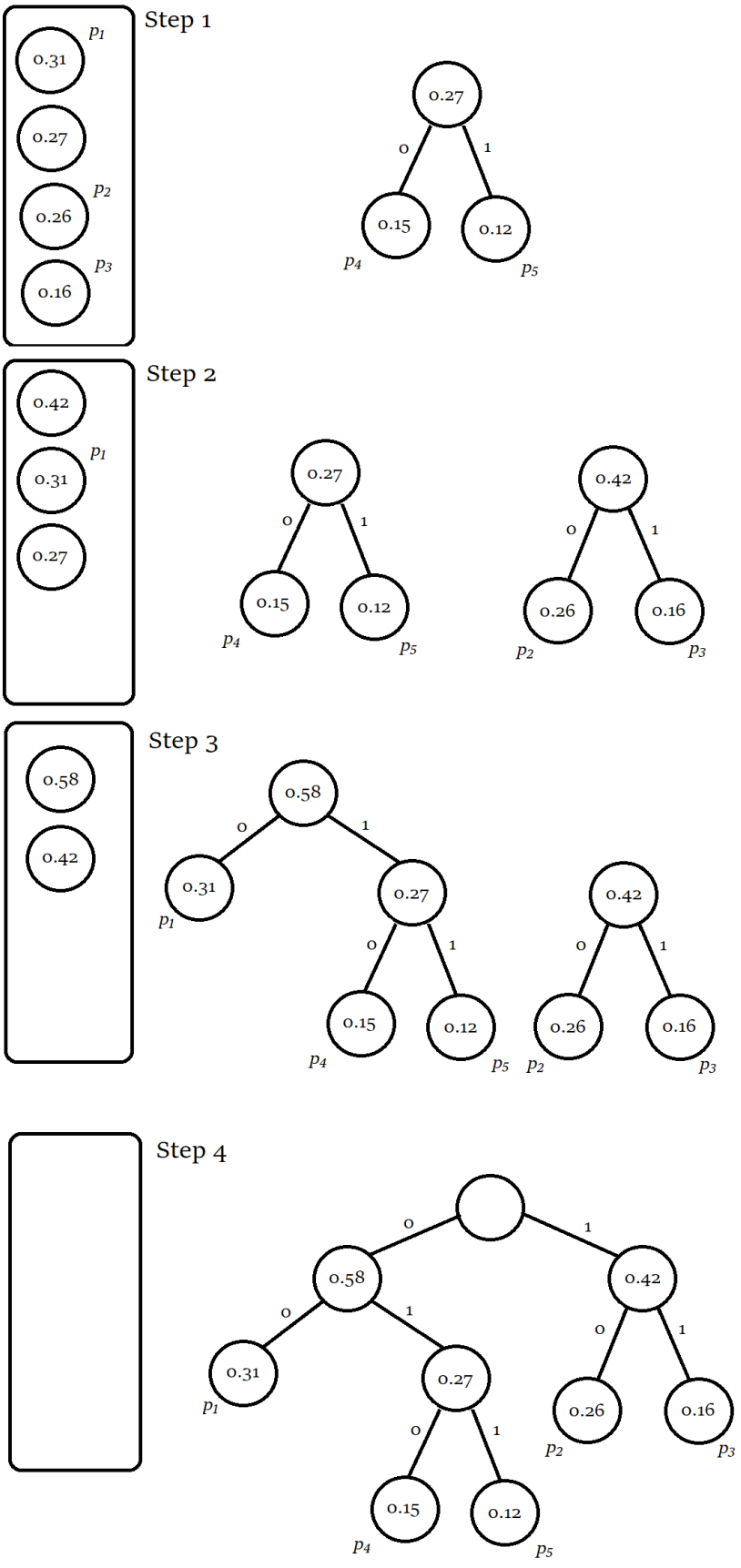


FIGURE 2.1: Step-by-step Huffman Coding. The box on the left is the forest. As there are no remaining nodes in the forest at step 4, the algorithm terminates.

2.1 k-RLL constrained Huffman codes

When we refer to constraining Huffman codes into a k-RLL code, we are actually mapping each Huffman generated codeword to a codeword that satisfies the k-RLL constraint. For the remainder of this section, all Huffman generated codewords will be referred to as *unconstrained* codewords.

Let m be the length of an unconstrained codeword and n to be the length of a word that satisfies the k-RLL constraint.

Let M and N be the set of unconstrained codewords and the set of all k-RLL codewords of length n. To preserve the property of uniquely decodable codes, we require that each unconstrained codeword be mapped to one and only one k-RLL codeword, making our mapping function injective. Therefore, $|M| \leq |N|$. Here, $|M|$ and $|N|$ refer to the size of sets M and N respectively.

Observe that since M only consists of unconstrained codewords, we have that $|M| = 2^m$. As for N, we refer to Immink's 'Codes for mass data storage systems' to determine the cardinality of this set [3]. We will state it in the following proposition.

Proposition 2.1.1 (Constraint Capacity)**.** The set of all k-RLL codewords of length n is $F_k(n)$, where

$$F_k(n) = \begin{cases} F_k(n-1) + F_k(n-2) + ... + F_k(n-k-1) & \text{if } n > k \\ 2^n & \text{if } n \leq k \end{cases}$$

Since we require $|M| \leq |N|$, we have that

$$\begin{aligned} 2^m &\leq F_k(n) \\ m &\leq \log_2 F_k(n) \\ \frac{n}{m} &\geq \frac{n}{\log_2 F_k(n)} \end{aligned}$$

Finally to preserve efficiency, we must choose n to be as small as possible, or m to be as large as possible. Thus

$$\frac{n}{m} = \frac{n}{\log_2 F_k(n)} \tag{2.1}$$

We define the above fraction to be the *expansion factor* for constraining the Huffman code into k-RLL constrained words. Let C_U and C_H be the average cost of the unconstrained code and the average cost of the constrained code respectively, we have the following

$$C_H = \frac{nC_U}{\log_2 F_k(n)} \tag{2.2}$$

It is clear that $C_H \geq C_U$, we are now concerned with whether the k-RLL constrained Huffman code preserves its optimality. In the next section, for certain probability distributions, we will see that this is not the case.

2.2 Golin's Tree Construction Algorithm for k-RLL codes

An alternative approach to Huffman's 2-step compression and encoding is to encode directly into a k-RLL code, thereby satisfying the k-RLL constraint. For this, we reference a dynamic programming algorithm to construct trees representing optimal codes proposed by Golin [1]. We will briefly describe the process as to how the trees are constructed, as the paper goes into more detail with their construction. First, we start with several definitions of trees.

Definition 2.2.1 (Depth, Level and Tree Fragments)**.** The *depth* of a node in a tree is the total cost of traversing from the root to that node. All nodes at depth k is defined to be the k-th *level* of a tree.

In a k-ary tree T, a *tree fragment* is a subtree consisting of a parent and k children. The *branch costs* of a tree fragment is a k-tuple of non-negative integer costs for each branch, from the leftmost branch to the rightmost branch.

A node in a tree that has at least one child is defined to be an *internal node*. Otherwise, it is a terminal node or *leaf*.

Example 2.2. In Figure 2.2, we construct a ternary tree made from $(1, 1, 3)$-cost tree fragments. For easier visualisation, the nodes are ordered by level and depth. The tree has two nodes at depth 1 and 2, three nodes at depth 3, 4 and 5, one node at depth 6 and 7. There are 5 internal nodes and 11 leaves.

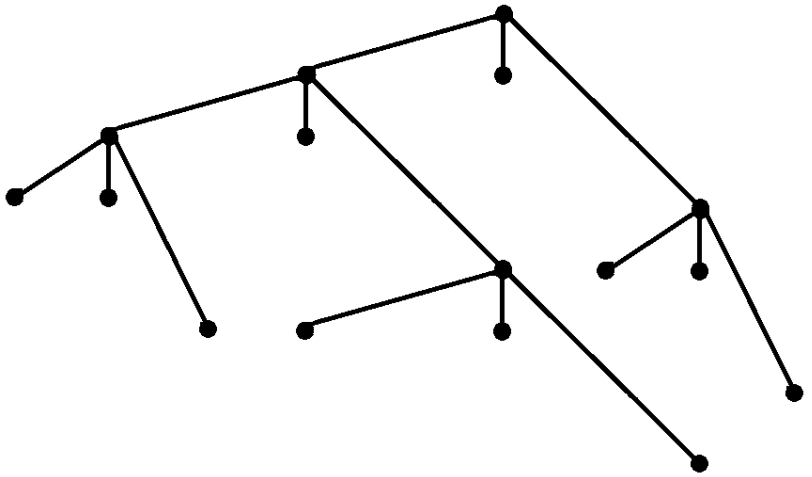


FIGURE 2.2: A ternary tree constructed from $(1, 1, 3)$-cost tree fragments.

Note that it seems pedantic to list nodes at each level, so we will introduce the concept of the so-called tree signature, as it defines a unique and compact method of describing the structure of the tree.

Definition 2.2.2 (Tree Signature)**.** Let T be a tree with t levels. An i-th level *signature*, $1 \leq i \leq t$ is the $k+1$-tuple

$$sig_i(T) = (u; l_1, l_2, ..., l_k)$$

where u is the number of leaves of depth less than i, and l_j is the number of nodes of depth $i + j$, $1 \leq j \leq k$.

Unlike Huffman coding, in which the tree is constructed bottom-up, Golin's dynamic programming algorithm constructs a tree top-down, appending tree fragments at each level. This means that we would start with a single tree fragment, with the remainder of the tree to be determined. Also, one can observe that the signature changes as we proceed down the levels of a tree. To rigorously construct the tree, we define the so-called tree expansion and the corresponding change in tree signature for each subsequent level.

Definition 2.2.3 (Tree and Signature Expansion)**.** Let T be a k-ary, i-level tree with signature $sig_i(T)$. The r-th tree expansion at level i is the tree T' constructed by appending r many tree fragments to leaves at depth i of T'.

A *signature expansion* is the mapping $sig_i(T) \to sig_{i+1}(T')$, such that

$$(u; l_1, l_2, ..., l_k) \mapsto (u + l_1; l_2, ..., l_k, 0) + r(-1; d_1, d_2, ...d_\Gamma)$$

where $(d_1, d_2, ..., d_\Gamma)$ is the *characteristic vector* of the branch cost, such that for $1 \leq \gamma \leq \Gamma$, d_γ is the number of c_i that is equal to γ.

Example 2.3. In figure 2.2, to reconstruct the tree via the expansion definition, we have to perform a 1-th expansion at each level of the tree. Starting at level 0 of the tree, the signature would evolve in the following way

$$(0; 2, 0, 1) \to (1; 2, 1, 1) \to (2; 3, 1, 1) \to (4; 3, 1, 1)$$

We can now formally explain how Golin's algorithm works.

1. Initialize a single k-ary Golin tree with single tree fragment having branch costs branches $(c_1, c_2, ..., c_k)$, set initial signature to be $(0; d_1, d_2, ..., d_\Gamma)$. Start tree at level 0.
2. While there are less than q leaves in each Golin tree:
   (a) Perform up to r tree and signature expansions, r is the number of leaves at the current level. Increase tree level by 1.
   (b) Determine the average cost of each expanded tree, assigning higher probabilities onto leaves of shallower depth and lower probabilities to leaves of deeper depth. Let C_G be the tree of lowest average cost.
   (c) If there is a new tree with lower average cost, update C_G and return to step 2. Otherwise, return to step 2.
3. Return C_G.

Example 2.4. In figure 2.3, we have $q = 4$ and $(c_1, c_2) = (1, 2)$. So we have the characteristic vector to be $(1, 1)$. Note that we can perform only 0 or 1 tree expansions in this toy example. A 0-th tree expansion essentially changes nothing in the tree, and we proceed down to the next level.

One can observe that traversing between signatures creates a directed acyclic graph, and thus each final construction of a Golin tree is unique when traversing from the initial signature to the final signature.

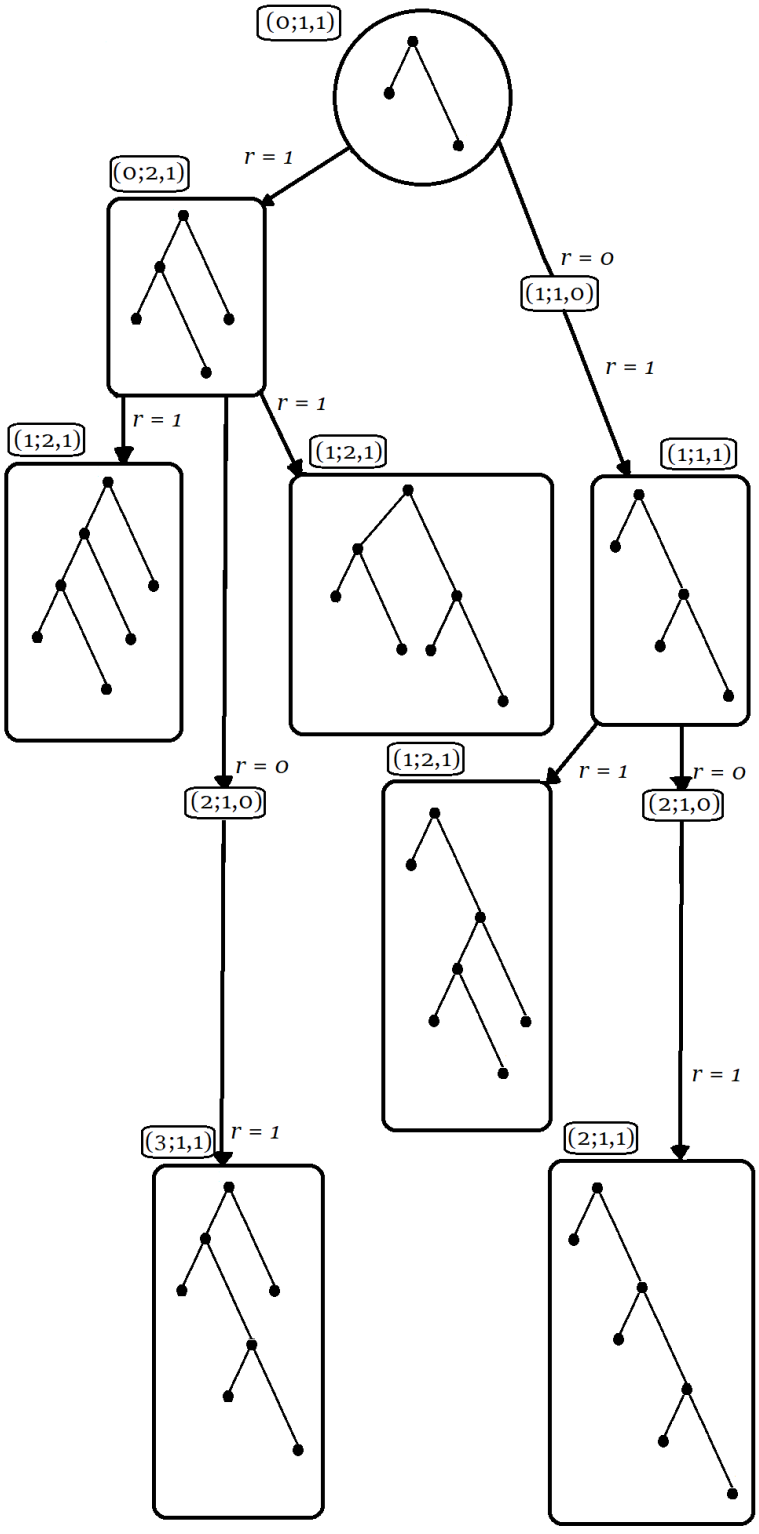


FIGURE 2.3: A toy example of tree and signature expansion for $q = 4$.

2.3 Comparison between Golin and Huffman over Zipf Distribution

When constructing a k-RLL code, we would construct $k+1$-ary trees, with the left branch having cost 1, the middle branch having cost 2 and so on until the right branch having cost $k+1$. With respect to Golin's algorithm, these trees would have signatures $(m; l_1, l_2, ..., l_k)$.

Before we begin a comparison proper, we first define the so-called Zipf Distribution.

Definition 2.3.1 (Zipf Distribution)**.** A discrete probability distribution $p_1, p_2, ..., p_q$ a real valued parameter $s > 1$ is *Zipf* if

$$p_i = \frac{i^{-s}}{\sum_{j=1}^{q} j^{-s}}, \; i = 1, ..., q$$

With this information in hand, we construct a tree for our code, following the Zipf distribution with differing values of s.

The following table are results of C_G and C_H, the average costs of the Golin's codes and the constrained Huffman codes respectively. We fix $q = 27$, which is the roman alphabet plus space. Note that we test for values of k up to 10, as the expansion factor approaches 1 for the constrained Huffman code, making no discernible difference between an unconstrained and constrained code. We also fix $n = 100$.

The results seem to imply that k-RLL constrained Huffman codes are less efficient when we restrict to lower k, namely for 2-RLL or 3-RLL codes. Indeed, this is when the expansion factor is at its largest. We also point out that the increase in s parameter also results in Golin's being more and more efficient. This implies that a more "steeper" probability distribution between each successive probability correlates to a lower average cost. In Chapter 3, we analyse the nature of such probability distributions, and find out how much they affect the resulting tree structure. Finally, we point out that as $k > 6$, Golin's construction method appears to lose its efficiency.

$s = 1.5$	C_G	C_H
$k = 1$	4.3667	**4.3495**
$k = 2$	3.4739	**3.4387**
$k = 3$	3.2019	**3.1955**
$k = 4$	3.1124	**3.1039**
$k = 5$	3.0839	**3.0644**
$k = 6$	3.0768	**3.0463**
$k = 7$	3.0997	**3.0376**
$k = 8$	3.1005	**3.0335**
$k = 9$	3.2014	**3.0314**
$k = 10$	3.2230	**3.0304**
$s = 2.0$	C_G	C_H
$k = 1$	**3.0147**	3.0829
$k = 2$	**2.4031**	2.4374
$k = 3$	2.2654	**2.2650**
$k = 4$	**2.1990**	2.2001
$k = 5$	2.1741	**2.1721**
$k = 6$	2.1664	**2.1592**
$k = 7$	2.1667	**2.1531**
$k = 8$	2.1647	**2.1502**
$k = 9$	2.1853	**2.1487**
$k = 10$	2.1894	**2.1480**
$s = 2.5$	C_G	C_H
$k = 1$	**2.1281**	2.3067
$k = 2$	**1.7768**	1.8236
$k = 3$	**1.6678**	1.6947
$k = 4$	**1.6362**	1.6461
$k = 5$	**1.6218**	1.6252
$k = 6$	**1.6154**	1.6156
$k = 7$	1.6139	**1.6110**
$k = 8$	1.6117	**1.6088**
$k = 9$	1.6153	**1.6077**
$k = 10$	1.6156	**1.6072**
$s = 3.0$	C_G	C_H
$k = 1$	**1.6280**	1.9041
$k = 2$	**1.4109**	1.5053
$k = 3$	**1.3625**	1.3989
$k = 4$	**1.3407**	1.3588
$k = 5$	**1.3332**	1.3415
$k = 6$	**1.3299**	1.3336
$k = 7$	**1.3287**	1.3298
$k = 8$	**1.3277**	1.3279
$k = 9$	1.3283	**1.3271**
$k = 10$	1.3281	**1.3266**

TABLE 2.1: Comparison of average costs between Golin and Huffman. The more efficient method is in bold for easier viewing.

Chapter 3

Analysis of Tree Constructions over Geometric Distribution

In this chapter we will analyse the tree structures of a 1-RLL code for small q, with assistance of Golin's tree construction algorithm. A 1-RLL code would generate binary trees with $(1, 2)$ cost branches, with the left branch and the right branch being encoded to 1 and 01 respectively.

3.1 Analysis for $q = 3, 4, 5$

Definition 3.1.1 (Geometric Distribution)**.** A discrete probability distribution $p_1, p_2, ..., p_q$ with a real valued parameter $0 < s < 1$ is *geometric* if

$$p_i = \frac{s^i}{\sum_{j=1}^{q} s^j}, \; i = 1, ..., q$$

It is clear that the probabilities will sum to one. We use a geometric distribution for our initial construction as it happens to have some nice properties.

We will state the following additional definitions for a binary tree.

Definition 3.1.2 (Appendages)**.** In a binary tree T, if we append another tree fragment to the left child of the previous tree fragment, we call that a *left appendage*, and a *right appendage* for the right child.

Note that it is also possible for a tree fragment to have both left and right appendages, and such cases will inevitably appear for increasing values of q and s. We will make such a distinction with the following definition.

Definition 3.1.3 (Sharp and Flat trees)**.** A tree is *sharp* if every tree fragment has at most one type of appendage. A tree is *flat* if at least one tree fragment has both types of appendages.

We will limit the focus of this thesis to sharp trees, as the properties of these trees will be readily proved later, while flat trees have a more complex structure.

A sharp binary tree can be recursively constructed top-down from a single tree fragment starting with a root and two children and we successively append a tree fragment to exactly one of its children, such that one child is a leaf and the other child becomes the next internal node, and so on until there are $q-1$ many tree fragments that correspond to q leaves.

We will start by looking at the two types of sharp trees below that look really special.

Definition 3.1.4 (Right and Left-leaning Trees)**.** A sharp tree is *right-leaning* if it only consists of right appendages, and *left-leaning* if it only consists of left appendages. A sharp tree that is neither right-leaning nor left-leaning is called an *intermediate tree*.

Definition 3.1.5 (Labelled Internal Node)**.** An internal node j , $1 \leq j \leq q-2$ in a sharp tree can be *labelled* such that exactly one child is the leaf p_j, and the last internal node $q-1$ having both p_{q-1} and p_q as its leaves.

For the main part of this section, we will now look at the structures of sharp trees for small values of q. For $q = 2$, there is only one trivial tree, a root and its two children.

Remark. If $q = 3$, there are exactly two possible trees: one right-leaning and one left-leaning.

For the remainder of this section, we shall ignore the case where two trees may be equally efficient, as it is trivial if we set the average cost of two trees to be equal.

The following proposition determines which of the two trees will be chosen.

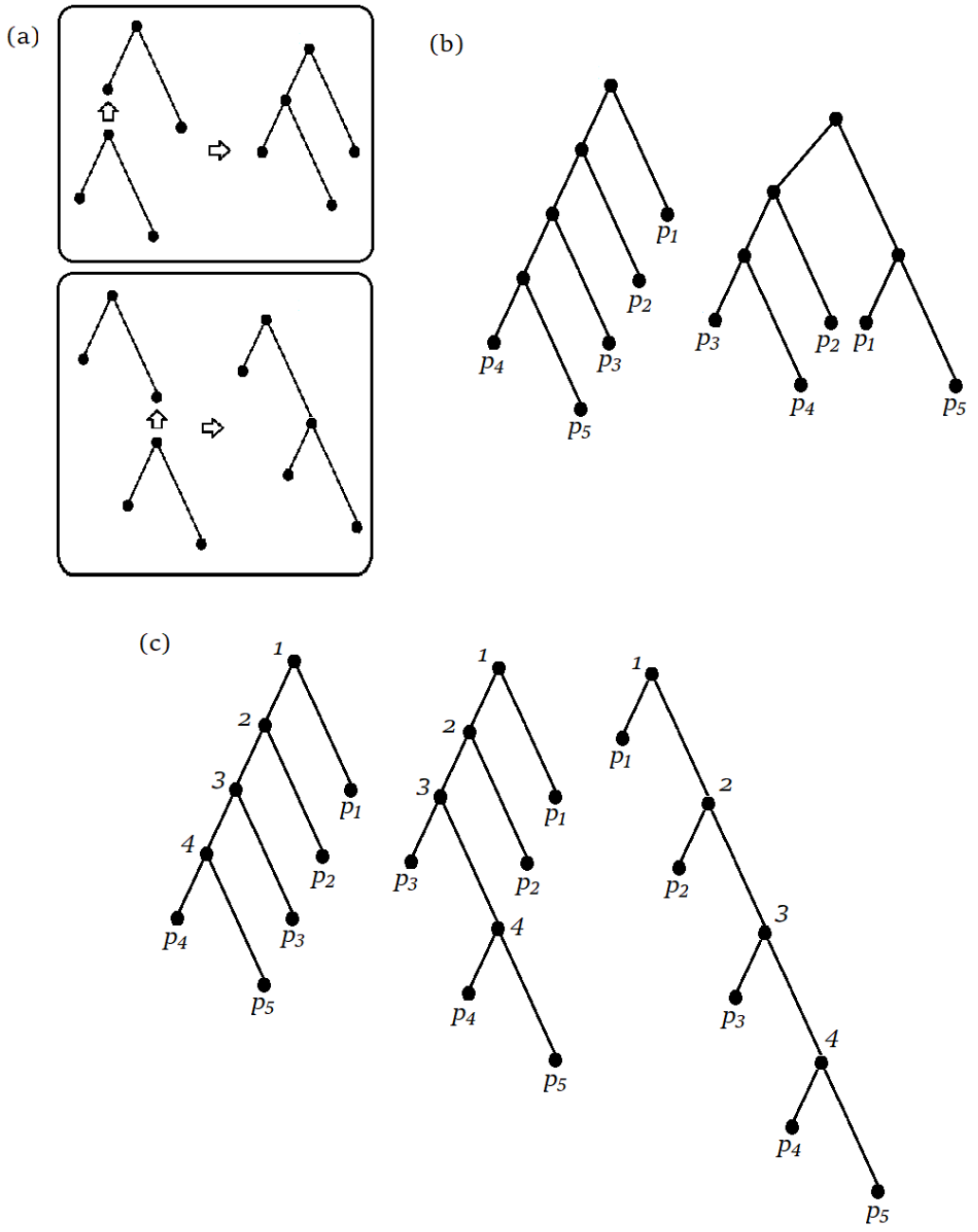


FIGURE 3.1: Binary trees with $(1, 2)$ cost branches. Part (a) shows a left appendage construction above and a right appendage below. Part (b) shows an example of a sharp tree and a flat tree for $q = 5$. Part (c) shows a left-leaning tree, an intermediate tree and a right-leaning tree on the left, middle and right for $q = 5$ respectively. The internal nodes are labelled accordingly.

Proposition 3.1.1 (Appendage Choice for $q = 3$)**.** Let $q = 3$. If p_1, p_2, p_3 follow a geometric distribution, we get the following dichotomy:

- If $p_1 > 0.5$, the right-leaning tree is optimal.
- If $p_1 < 0.5$, the left-leaning tree is optimal.

Alternatively, with respect to the distribution parameter $0 < s < 1$:

- If $s < \Psi_2$, the right-leaning tree is optimal.
- If $s > \Psi_2$, the left-leaning tree is optimal.

Note that

$$\Psi_2 = \phi^{-1} = \left(\frac{1+\sqrt{5}}{2}\right)^{-1} = \left(\frac{\sqrt{5}-1}{2}\right) \approx 0.618...$$

is the unique positive root $\Psi_2^2 = 1 - \Psi_2$, otherwise known as the reciprocal of the golden ratio.

Proof. Let the average costs of the left-leaning and right-leaning tree be T_l and T_r respectively. By simply drawing out the two trees, we have that

$$T_r = p_1 + 3p_2 + 4p_3 \tag{3.1}$$

$$T_l = 2p_1 + 2p_2 + 3p_3 \tag{3.2}$$

Note that p_1, p_2, p_3 are geometrically distributed, with parameter s as the constant ratio between pairs of successive probabilities, as follows

$$s = \frac{p_j}{p_{j+1}}, \, 1 \leq j \leq q \tag{3.3}$$

Thus we can rewrite (3.1) and (3.2) as the following

$$T_r = p_1 + 3p_1 s + 4p_1 s^2 \tag{3.4}$$

$$T_l = 2p_1 + 2p_1 s + 3p_1 s^2 \tag{3.5}$$

Assume the right-leaning tree is more efficient than the left-leaning tree, we get the inequality

$$p_1 + 3p_1s + 4p_1s^2 < 2p_1 + 2p_1s + 3p_1s^2 \tag{3.6}$$

Observe that $1 - p_1 = p_2 + p_3$, which can rewritten as $1 - p_1 = p_1s + p_1s^2$. We can then substitute this into (3.6)

$$\begin{aligned} p_1 + 3(p_1s + p_1s^2) + p_1s^2 &< 2p_1 + 2(p_1s + 2p_1s^2) + p_1s^2 \\ p_1 + 3(1 - p_1) + p_1s^2 &< 2p_1 + 2(1 - p_1) + p_1s^2 \\ -2p_1 + 3 &< 2 \\ -2p_1 &< -1 \\ p_1 &> 0.5 \end{aligned}$$

The proof for the remainder of the trichotomy is equally similar.

Using equation (3.6), we can also determine s

$$\begin{aligned} p_1 + 3p_1s + 4p_1s^2 &< 2p_1 + 2p_1s + 3p_1s^2 \\ p_1s + p_1s^2 &< p_1 \\ s + s^2 &< 1 \\ s^2 &< 1 - s \\ s &< \Psi_2 \end{aligned}$$

□

We can determine the choice of appendages for the case $q = 4$ with a similar method, this time noting that there are more than two trees. In fact, there are a total of five distinct trees if we exhaust all possible constructions using tree fragments and appendages. For this, we introduce the notion of equivalent trees.

Definition 3.1.6. (Equivalent Trees) Two trees are *equivalent* if the cost of traversing each leaf in both trees are identical.

Example 3.1. In Figure 3.2, we see all possible constructions of trees for $q = 4$. Tree 1 is left-leaning, tree 5 is right-leaning. All other trees in between (with the

exception of tree 2) are intermediate trees. Note that trees 1 and 2 are equivalent. Therefore, all trees in $q = 4$ are sharp.

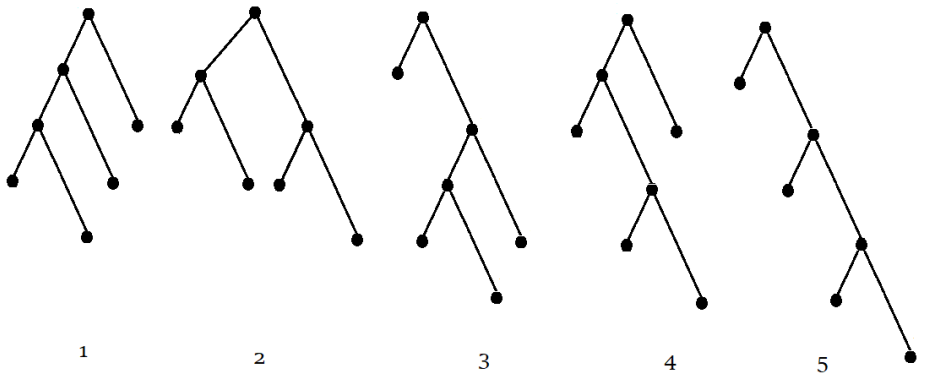


FIGURE 3.2: All possible trees that can be constructed with $q = 4$.

Finally, we define a tree to be *inaccessible* if it cannot be optimal for all real $0 < s < 1$.

Remark. If $q = 4$, there are a total of four possible trees up to equivalence: A right-leaning tree, two intermediate trees and a left-leaning tree.

Below we have an analogue of Proposition 3.1.1 for the case $q = 4$ to determine the appendage choice.

Proposition 3.1.2 (Appendage Choice for $q = 4$)**.** Let $q = 4$. If p_1, p_2, p_3, p_4 follow a geometric distribution, we get the following trichotomy:

- If $p_1 > 0.5$, tree 5, the right-leaning tree is optimal.
- If $p_1 < 0.5$ and $p_2 > 0.5(1 - p_1)$, tree 4, one of the intermediate trees, is optimal.
- If $p_2 < 0.5(1 - p_1)$, tree 1, the left-leaning tree is optimal.

Similarly, with respect to the distribution parameter $0 < s < 1$:

- If $s < \Psi_3$, tree 5, the right-leaning tree is optimal.
- If $\Psi_3 < s < \Psi_2$, tree 4, one of the intermediate trees, is optimal.

- If $s > \Psi_2$, tree 1, the left-leaning tree is optimal.

where Ψ_3 is the unique positive root such that $\Psi_3^3 = 1 - \Psi_3 - \Psi_3^2$.

Proof. The first case of the trichotomy is similar to the previous proposition, so it remains to prove the second case. We first show that one of the trees is inaccessible. This is tree 3 in Figure 3.2. Assume that tree 3 is optimal, this means it has to be more efficient than all the other possible trees. Comparing trees 3 and 4, we get

$$\begin{aligned} p_1 + 4p_1s + 4p_1s^2 + 5p_1s^3 &< 2p_1 + 2p_1s + 4p_1s^2 + 5p_1s^3 \\ 2p_1s &< p_1 \\ s &< 0.5 \end{aligned}$$

However, by comparing trees 3 and 5

$$\begin{aligned} p_1 + 4p_1s + 4p_1s^2 + 5p_1s^3 &< p_1 + 3p_1s + 5p_1s^2 + 6p_1s^3 \\ p_1s &< p_1s^2 + p_1s^3 \\ p_1 &< p_1s + p_1s^2 \\ 1 &< s + s^2 \\ s &> \Psi_2 \end{aligned}$$

we get contradicting values of s. We are thus left with trees 1, 4 and 5 as possible trees for $q = 4$.

If we repeat the same steps of the previous proof, one can arrive with a similar inequality. Comparing trees 4 and 5,

$$\begin{aligned} 2p_1 + 2p_1s + 4p_1s^2 + 5p_1s^3 &< p_1 + 3p_1s + 5p_1s^2 + 6p_1s^3 \\ p_1 + 2(1 - p_1) + 2p_1s^2 + 3p_1s^3 &< 3(1 - p_1) + 2p_1s^2 + 3p_1s^3 \\ p_1 &< 1 - p_1 \\ p_1 &< 0.5 \end{aligned}$$

and comparing trees 1 and 4,

$$\begin{aligned} 2p_1 + 2p_1s + 4p_1s^2 + 5p_1s^3 &< 2p_1 + 3p_1s + 3p_1s^2 + 4p_1s^3 \\ 2(1-p_1) + 2p_1s^2 + 3p_1s^3 &< 3(1-p_1) + p_1s^3 \\ 2p_1s^2 + 2p_1s^3 &< 1 - p_1 \\ 2(1-p_1) - 2p_1s &< 1 - p_1 \\ 2p_1s &> 1 - p_1 \\ p_1s &> 0.5(1-p_1) \end{aligned}$$

where $p_2 = p_1 s$.

We can also determine s the same way as follows

$$\begin{aligned} 2p_1 + 2p_1s + 4p_1s^2 + 5p_1s^3 &< p_1 + 3p_1s + 5p_1s^2 + 6p_1s^3 \\ 2 + 2s + 4s^2 + 5s^3 &< 1 + 3s + 5s^2 + 6s^3 \\ 1 &< s + s^2 + s^3 \\ s &> \Psi_3 \end{aligned}$$

$$\begin{aligned} 2p_1 + 2p_1s + 4p_1s^2 + 5p_1s^3 &< 2p_1 + 3p_1s + 3p_1s^2 + 4p_1s^3 \\ 2s + 4s^2 + 5s^3 &< 3s + 3s^2 + 4s^3 \\ s^2 + s^3 &< s \\ s + s^2 &< 1 \\ s &< \Psi_2 \end{aligned}$$

□

We now look at the last case $q = 5$, which contains more complicated tree structures. As we saw in Figure 3.1 part (b), a flat tree exists as a possible construction of an optimal tree. We look at the following example.

Example 3.2. In Figure 3.3, we see all twelve possible constructions of trees for $q = 5$. Tree 1 is the flat tree, tree 2 is left-leaning, trees 3 and 4 are intermediate trees and tree 5 is the right-leaning trees. Trees 2a and 2b are equivalent to tree

2, and tree 3a is equivalent to tree 3. Finally, the four trees at the bottom are inaccessible.

Since it is not that discernible at lower q, we explain the reason for the bottom inaccessible trees in the next section. First, we conclude this section by stating the following proposition below without proof, as it is identical to the above two propositions.

Proposition 3.1.3 (Appendage Choice for $q = 5$)**.** Let $q = 5$. If p_1, p_2, p_3, p_4, p_5 follow a geometric distribution, with respect to the s parameter, we get the following cases:

- If $s < \Psi_4$, tree 5, the right-leaning tree is optimal.
- If $\Psi_4 < s < \Psi_3$, the intermediate tree 4 is optimal.
- If $\Psi_3 < s < \Psi_2$, the intermediate tree 3 is optimal.
- If $\Psi_2 < s < \sqrt{\Psi_2}$, tree 2, the left-leaning tree is optimal.
- If $s > \sqrt{\Psi_2}$, tree 1, the flat tree is optimal.

  Note that $\sqrt{\Psi_2}$ is the unique positive root of $\Psi_2^4 = 1 - \Psi_2^2$.

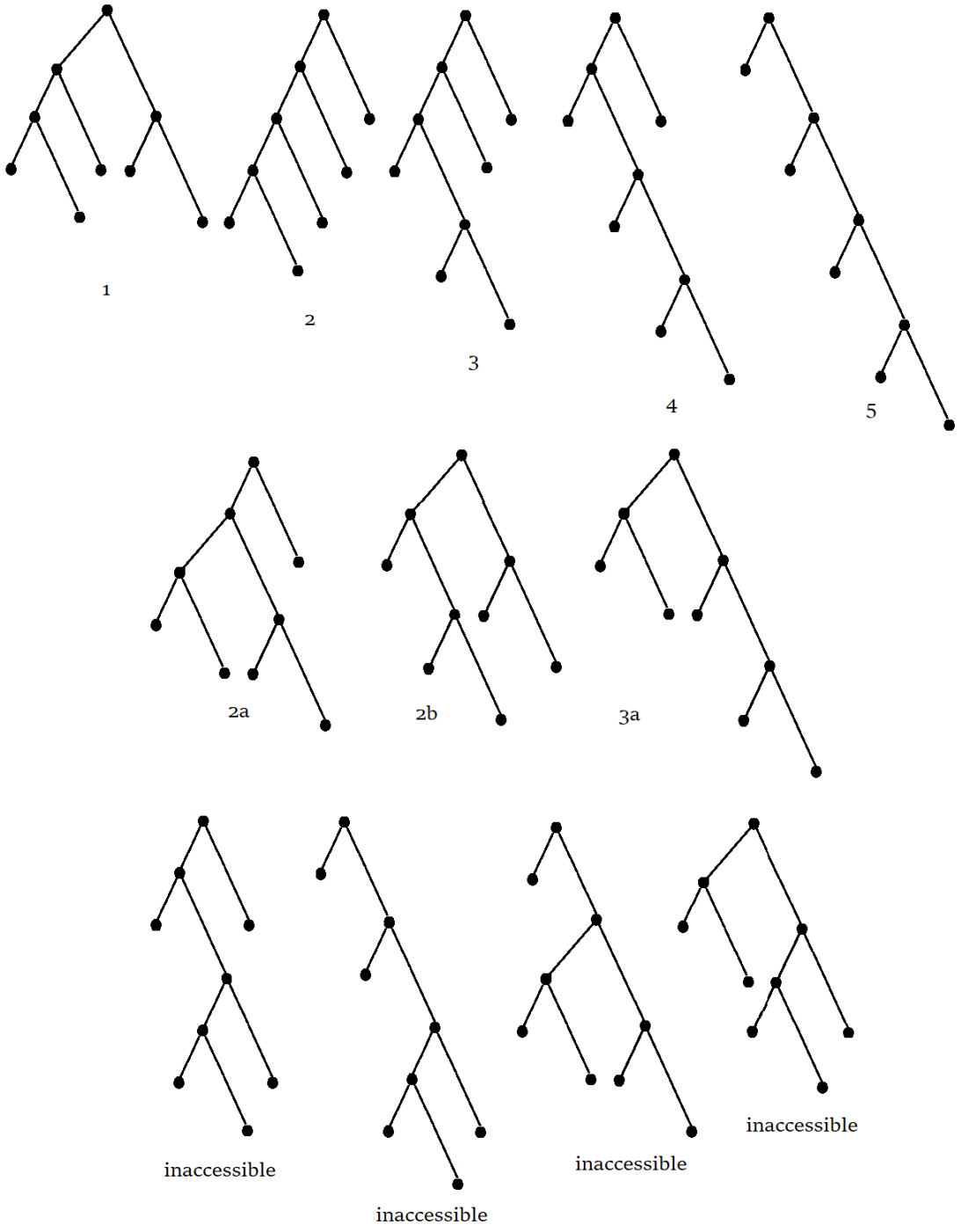


FIGURE 3.3: All possible trees that can be constructed with $q = 5$.

3.2 Tree constructions for higher q

Observing the cases $q = 3, 4, 5$, we can determine a particular pattern when deciding between tree is chosen to be optimal: s is the unique positive root of a specific polynomial. This polynomial remains consistent for sharp trees, but it gets progressively complicated when flat trees are considered optimal for higher values of s.

We can, however generalise the pattern for sharp trees. We first require two definitions to easier visualise this.

Definition 3.2.1. The k-th intermediate tree is the sharp tree construction that starts with k left appendages, thereafter succeeded by right appendages.

We say that the s parameter *partitions* two trees if for a real valued integer $0 < t < 1$, one tree is more optimal than the other when $s < t$ and vice versa if $s > t$.

Example 3.3. Tree 4 in Figures 3.2 and 3.3 are the first intermediate trees for $q = 4$ and $q = 5$. Furthermore, all partitions of sharp trees are unique positive roots of a polynomial.

The following theorem shows the interesting structure of the aforementioned polynomials.

Theorem 3.2.1 (Generalisation of s parameter for higher q)**.** Let $p_1, p_2, ..., p_q$ follow a geometric distribution. If $s < \Psi_{q-1}$, we have that the right-leaning tree optimal. Furthermore, each s that partitions each successive sharp tree, up to the left-leaning tree, is $\Psi_{q-2}, ..., \Psi_3, \Psi_2$ respectively, such that

$$0.5 < \Psi_{q-1} < \Psi_{q-2} < ... < \Psi_3 < \Psi_2$$

Ψ_{q-1} here is the unique positive root such that

$$\Psi_{q-1}^{q-1} = 1 - \Psi_{q-1} - \Psi_{q-1}^2 - ... - \Psi_{q-1}^{q-2}$$

Proof. Let T_r and T_i be the average cost of the right-leaning tree and the first intermediate tree respectively. Again, by drawing out the two trees, we have the

following general formulas

$$T_r = p_1 + \sum_{k=2}^{q-1}(2k-1)p_k + (2q-2)p_q \tag{3.7}$$

$$T_i = 2p_1 + 2p_2 + \sum_{k=3}^{q-1}(2k-2)p_k + (2q-3)p_q \tag{3.8}$$

Now assume the right leaning tree is more efficient than the first intermediate tree we get

$$\begin{aligned}
p_1 + \sum_{k=2}^{q-1}(2k-1)p_k + (2q-2)p_q &< 2p_1 + 2p_2 + \sum_{k=3}^{q-1}(2k-2)p_k + (2q-3)p_q \\
p_2 + p_3 + ... + p_q &< p_1 \\
p_1 s + p_1 s^2 + ... + p_1 s^{q-1} &< p_1 \\
s + s^2 + ... + s^{q-1} &< 1 \\
s^{q-1} &< 1 - s - s^2 - ... - s^{q-2} \\
s &< \Psi_{q-1}
\end{aligned}$$

Now, it is clear that $\Psi_{q-2}, ..., \Psi_3, \Psi_2$ partition each sharp tree, as this can be shown by recursively appending a tree fragment to the leaf of deepest depth. It thus remains to show that the chain of inequalities is true. Consider the polynomial

$$P_{q-1}(x) = 1 - x - x^2 - ... - x^{q-1} \tag{3.9}$$

Then Ψ_{q-1} is the root of $P_{q-1}(x)$. Note that $P_{q-1}(1) < 0$ and $P_{q-1}(0.5) > 0$, and since Ψ_{q-1} is a positive root, we get

$$0.5 < \Psi_{q-1} < 1 \tag{3.10}$$

Likewise, we can see that

$$P_{q-2}(x) = 1 - x - x^2 - ... - x^{q-2} \tag{3.11}$$

has the root Ψ_{q-2}. Comparing equation (3.9) and (3.11), we have that

$$P_{q-1}(x) = P_{q-2}(x) - x^{q-1} \tag{3.12}$$

and $x^{q-1} > 0$ for positive roots, thus we get the inequality

$$\Psi_{q-1} < \Psi_{q-2}$$

Proving the remainder of the chain of inequalities is similar, completing the proof. □

We can now explain why the bottom trees in Figure 3.3 were inaccessible. One can intuitively see that once we attach a right appendage to the previous tree fragment, it is no longer possible to attach a left appendage to the successive tree fragments. This is can be shown in the following corollary.

Corollary 3.2.1.1 (Sharp Tree inaccessibility for a Geometric Distribution)**.** If $p_1, p_2, ..., p_q$ follow a geometric distribution and $p_1 > 0.5$, all tree fragments must be right appendages and the resulting sharp tree has to be a right-leaning tree. Furthermore,

Proof. Assume that $p_1 > 0.5$ but $p_2 < 0.5(1-p_1)$, this would mean we would assign a right appendage to the first tree fragment followed by a left appendage to the next tree fragment. We get the following derivation

$$\begin{aligned} p_1 s &< 0.5(1 - p_1) \\ p_1 s &< 0.5 - 0.5p_1 \\ p_1 s + 0.5p_1 &< 0.5 \\ p_1 &< \frac{0.5}{s + 0.5} \\ 0.5 &< \frac{0.5}{s + 0.5} \\ 0.5s + 0.25 &< 0.5 \\ s &< 0.5 \end{aligned}$$

However, we have that $s < 0.5 < \Psi_{q-1}$ which is a contradiction to Theorem 3.2.1 as this would result in a right-leaning tree. □

We will now focus the remainder of this section on flat trees appearing for higher q.

Unlike sharp trees, the general pattern of which flat tree is chosen is currently not known as there are many tedious choices. However, we have empirically found that the partition between the last sharp tree, the left-leaning tree, and the first flat tree appear to follow a general formula. Furthermore, the first flat tree appears to follow a particular shape. This can be better visualised in Figure 3.4. The results of the parameter partitions are also summarised in Table 3.1 for $q = 6, 7, 8$.

Note that the first flat tree F_1 appears to have a left and right appendage on the first fragment. It is currently not known if this is the case for $q > 8$.

As the trees appear flatter, we note that it corresponds to the probability distribution being more "gentle". When the probabilities are equal, the average costs of such optimal trees are studied by Horibe [4].

q	s Partitions Between Pairs of Trees			
	Left-leaning Tree, F_1	F_1, F_2	F_2, F_3	F_3, F_4
$q = 6$	$s \approx 0.7549$ $s^5 = 1 - s^4 - s^3$	–	–	–
$q = 7$	$s \approx 0.7245$ $s^6 = 1 - s^5 - s^4 - s^3$	$s \approx 0.8376$ $s^5 = 1 - s^3$	–	–
$q = 8$	$s \approx 0.7081$ $s^7 = 1 - s^6 - s^5 - s^4 - s^3$	$s \approx 0.7862$ $s^4 = 1 - s^2$	$s \approx 0.8087$ $s^5 = 1 - s^2$	$s \approx 0.8785$ $s^7 = 1 - s^4$

TABLE 3.1: Optimal flat trees and their corresponding partitions s between each tree for $q = 6, 7, 8$. The corresponding flat trees are given in Figure 3.4.

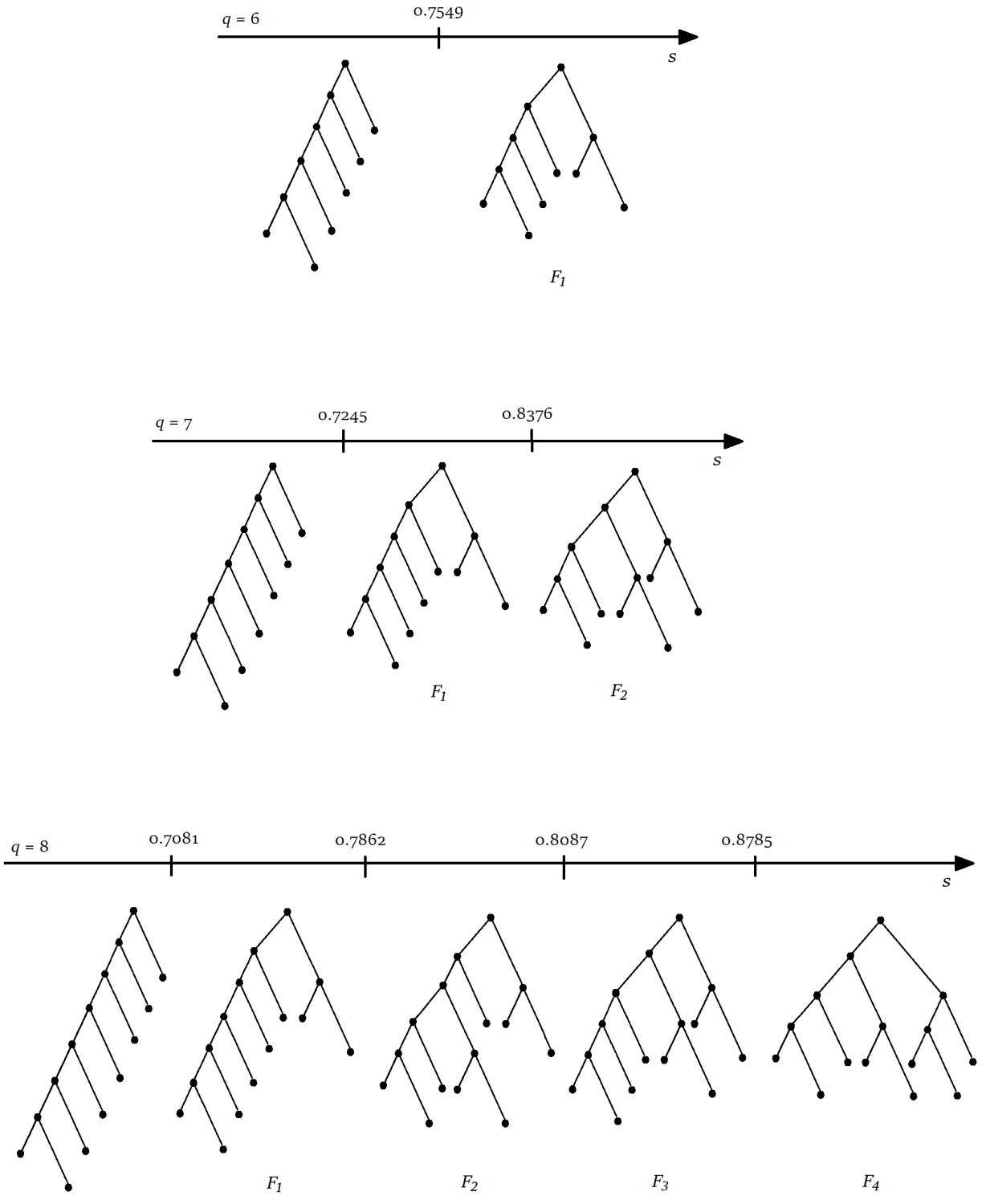


FIGURE 3.4: Optimal Flat Trees for $q = 6, 7, 8$, ordered by increasing values of s.

Chapter 4

Conclusion and Future Work

In Chapter 2, we found the construction of Golin's methods are largely more efficient than k-RLL constrained Huffman codes. However, as the results are mostly narrowed down to a small set of fixed constraints, the overall efficiency of Golin's method may be disputed.

Other probability distributions may be applied, such as the Geometric distribution in Chapter 3, or higher values of q may be used. It is worthy to point out that Golin's dynamic programming algorithm has a running time of $O(q^{k+2})$ [1], and for relatively large $q > 75$, the algorithm may take hours to complete on a standard PC. Another paper by Golin promises a linear-time approximation for Huffman Coding with letter costs, which might prove to be easier to approximate the construction of such large trees [5].

In Chapter 3, we managed to classify the structures of sharp trees for a 1-RLL code over a Geometric distribution. It is however, currently unknown what the structure of a k-RLL code would look like over a Geometric distribution for $k > 1$. We predict the structure to have similar properties to λ-weight balancing, which is mentioned by Horibe [4].

Similarly, one can look at Manickam, where they have gone in great detail on the optimality for $(k-1, k)$-RLL constraints. Here, the notation refers to the encoding alphabet having a minimum of $k-1$ run of zeroes to a maximum of k zeroes [2].

We may also wish to explore different probability distributions over the same 1-RLL code. We have found, empirically, that the Zipf distribution does not correspond to

the same sharp tree pattern as the Geometric distribution when $q > 7$, no matter how "steep" we set the distribution. We speculate that the ratio between pairs of probabilities significantly affect the tree structure.

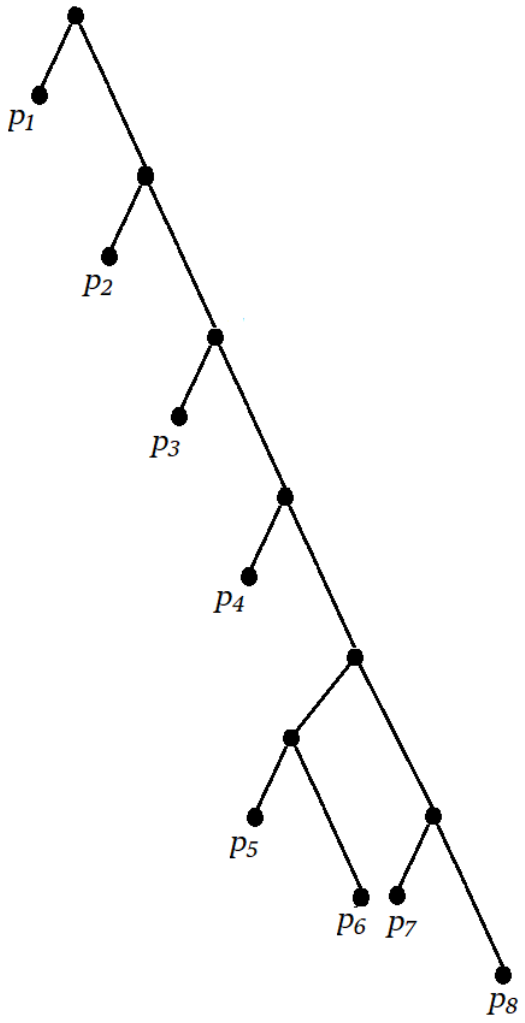


FIGURE 4.1: Tree with Zipf distribution for $q = 8$

Bibliography

[1] Mordecai Golin and Günter Rote. A dynamic programming algorithm for constructing optimal prefix-free codes for unequal letter costs. pages 256–267, 07 1995. doi: 10.1007/3-540-60084-1_79. 2, 11, 32

[2] Shivkumar Manickam and Navin Kashyap. On minimum expected length prefix codes satisfying a (d, k) runlength-limited constraint. pages 648–652, 10 2018. doi: 10.23919/ISITA.2018.8664350. 4, 32

[3] Kees A. Schouhamer. Immink. *Codes for mass data storage systems*. Shannon Foundation Publisher, 2004. 10

[4] Y. Horibe. An entropy view of fibonacci trees. *Fibonacci Quart.*, 20(2):168–178, 1982. URL `www.scopus.com`. Cited By :10. 30, 32

[5] Mordecai Golin, Claire Mathieu, and Neal Young. Huffman coding with letter costs: A linear-time approximation scheme. *SIAM Journal on Computing*, 41, 05 2002. doi: 10.1137/100794092. 32