ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

UNA CODIFICACIÓN PREFIJA PARA UN IDIOMA ARTIFICIAL
A PREFIX ENCODING FOR A CONSTRUCTED LANGUAGE

Realizado por
José Antonio Ortega Toro
Tutorizado por
Francisco José Vico Vela
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2015

Fecha defensa:
El Secretario del Tribunal

Resumen: Este trabajo se centra en el estudio formal y técnico de algunos componentes de un lenguaje artificial.

Como primera parte del trabajo, se estudiará una posible codificación para el lenguaje, haciendo énfasis en que se trate de una codificación libre de prefijo, para lo cuál se realizará una extensión del algoritmo de Huffman de binario a n-ario.

Debido a que en el lenguaje no podemos conocer a priori la frecuencia de uso de las palabras, se hará un estudio y se propondrán diversas estrategias para un sistema de palabras abierto, analizando previamente el número de palabras existentes en lenguajes naturales actuales.

Como posible mejora a la codificación, también analizaremos el problema de la pérdida de sincronización, así como su solución: la auto sincronización (self-synchronizing codes), un estudio de los t-codes con el número de palabras posibles para el lenguaje, así como otras alternativas.

Por último, y desde un apartado menos formal y más técnico, se han desarrollado diversas aplicaciones para la implementación del lenguaje: un sintetizador de voz, un reconocedor de voz y una fuente para el uso del lenguaje en procesadores de texto. Para cada uno de estos desarrollos se detallará el proceso seguido para su construcción, así como los problemas encontrados o por resolver en cada uno de ellos.

Palabras claves: teoría de la codificación, lenguaje artificial, teoría de la información, códigos libres de prefijo, codificación de Huffman, reconocimiento de voz, síntesis de voz.

Abstract: This work focuses in the formal and technical analysis of some aspects of a constructed language.

As a first part of the work, a possible coding for the language will be studied, emphasizing the prefix coding, for which an extension of the Huffman algorithm from binary to n-ary will be implemented.

Because of that in the language we can't know a priori the frequency of use of the words, a study will be done and several strategies will be proposed for an open words system, analyzing previously the existing number of words in current natural languages.

As a possible upgrade of the coding, we'll take also a look to the synchronization loss problem, as well as to its solution: the self-synchronization, a t-codes study with the number of possible words for the language, as well as other alternatives.

Finally, and from a less formal approach, several applications for the language have been developed: A voice synthesizer, a speech recognition system and a system font for the use of the language in text processors. For each of these applications, the process used for its construction, as well as the problems encountered and still to solve in each will be detailed.

Keywords: coding theory, constructed language, information theory, prefix code, huffman coding, speech recognition, voice synthesization.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Since more than half of a century we have been dealing with the automatic processing of natural languages, in the beginning for the "simple" task of automatic translation of texts to English. When we realized that the objective was not so easy to achieve, computational linguistics was born as a new field of study, with the objective of developing algorithms and software that intelligently process natural languages. In the 60s, with the arrival of the artificial intelligence, the field of computational linguistics became a sub-field of it, dealing with the comprehension at human-level.

In most of the field, computational linguistics focuses on the processing or the modeling of natural languages, but in this case we're going to see another of its applications: The modeling of constructed languages.

We could say that constructed languages can be divided into three fundamental groups (regardless of whether they have a computational origin or not): The languages that are aimed to make the communication easier between people (auxiliary languages, e.g. esperanto), the ones that are born with a creative purpose (artistic languages, e.g. klingon), and finally the engineered languages. In this last group we can distinguish two types: The languages based on some kind of idea (philosophical languages) and the languages created for the purpose of experimentation (experimental languages, like we'll see here).

As a base for a language that we are going to build, we have the vocabulary, which will be defined according to a coding. Providing to this coding some of the properties that can help us to resolve actual problems in natural languages, or at least can reduce its complexity. They can also (using existing techniques to build the vocabulary) remove some non-desirable features of the natural languages.

These properties and techniques we have been talking about are from the coding theory, which appeared in 1948 with the publication of "A Mathematical Theory of Communication" by

Claude Shannon[1]. Since then, the field has been growing up, introducing new concepts and techniques that today are still used on several science fields (electrical engineering, mathematics, computer science, information theory...) in order to obtain different ways for transmitting data, in a reliable and efficient way.

One key feature that can be achieved applying the coding theory knowledge and that will be one of the main objectives of this work is the prefix coding, resulting in uniquely decodable codes, which we'll talk later. One of the most famous codes of this kind are the Huffman codes, introduced in 1952 by David A. Huffman on his paper "A Method for the Construction of Minimum-Redundancy Codes"[2].

Finally, once defined the lexicon of the language through the encoding, we can proceed with other subtasks of the computational linguistics: voice synthesization and speech recognition, two of the big problems in natural language processing. Dealing with a constructed language built with different features from a natural language in its lexicon, we can analyze if these problems are easier to deal with.

So, the objectives of this work are: first, study and propose a coding scheme for the Unilan language that will have at least the prefix free property, analyze the resulting words system distribution and propose some alternatives for a self-synchronizing codification.

And secondly, develop a first version of a voice synthesizer, a speech recognition system and a system font for Unilan, and analyze the resultant developments for further work.

The following technologies will be used in this work:

Python will be used in general for coding algorithms, STT (Speech-to-text) and TTS (Text-to-speech), as well as CUDA for the STT first approach and the MBROLA speech synthesizer and MBROLA voices from the MBROLA project for the TTS.

For the system font development, the FontForge open source software will be used.

Finally, for the study about the performance of the STT engine, a website has been developed using the Django web framework (+MySQL for the data collection).

And the structure of the work will be the following:

- First, we'll introduce Unilan, the constructed language that we'll work with and is currently on development, as well as its goals, alphabet, vocabulary and phonetics.
- Secondly, we'll move on to the theoretical study, that is, the introduction of the different concepts related to the coding theory, analysis of the different techniques for achieve some advantages on the coding, adaptation and implementation of some existing techniques to the Unilan language, and lastly, discussion of the results achieved.

In first place we'll discuss about prefix coding, and then we'll move on to the synchronization loss problem.

- The last part of the work will be focused on the applications we mentioned:
  - First we'll deal with the voice synthesization (Text-to-speech, TTS).
  - Then, we'll move to the speech recognition system (Speech-to-text, STT).
  - To end, the system font for the Unilan language.

Chapter 2

Unilan

2.1 Introduction

Unilan is a pidgin artificial language that intends to become a natural language by the interaction of humans with machines. Different from a creole language, Unilan does not inherit its vocabulary from any phylogenetic parent language (it is, an a priori language), instead, it will be created by an algorithm as new words are defined within Unilan's semantics.

Since a pidgin language is a fundamentally simple form of communication, the grammar and phonology are usually as simple as possible, and they consist of:

- Uncomplicated clausal structure (e.g., no subordinate clauses, etc.)
- Reduction or elimination of syllable codas
- Reduction of consonant clusters or breaking them with epenthesis
- Basic vowels, such as [a, e, i, o, u]
- No tones, such as those found in West African and Asian languages
- Use of separate words to indicate tense, usually preceding the verb
- Use of reduplication to represent plurals, superlatives, and other parts of speech that represent the concept being increased
- A lack of morphophonemic variation

2.2 Problems solved by Unilan

- 1 glyph $\rightarrow$ 1 letter
  - Allography: There is no distinction between uppercase and lowercase (or other glyphs).
- 1 glyph $\rightarrow$ 1 digit
  - Phonetics of numbers.
- 1 morpheme $\rightarrow$ 1 phoneme
  - There is no morphophonemic variation, each symbol represents a different syllable, and there are no allophones (as proposed in the Shavian alphabet).
- 1 word $\rightarrow$ 1 concept
  - Homonymy: A word cannot have multiple unrelated meanings.
  - Polysemy: A word cannot have multiple related meanings.
- Computer language = natural language
- Semantic network
  - Resolving the semantics ambiguities.
- Centralized words generator
  - This avoids the phonetic, syntactic and semantic drift.
- Sign language, since every symbol can be represented with the two hands (one for the consonant, the other for the vowel), sentences shortens with respect to human languages, and there are no interruptions in the communication.

2.3 Alphabet

Each letter is an open syllable with one, and only one onset consonant, and it results from a typographic ligature of a consonant and a vowel (always in this order) of a subset of the latin alphabet. 16 consonants and the 5 vowels are used, yielding an 80-character set.

Alphabetical order is determined, firstly, by the order of occurrence of the consonant in the latin alphabet, and, secondly, by the order of the vowels (ba, be, bi, bo, bu, ca, ..., zu).

Here is the latest version of the letters used and its binary representation:

TABLE 2.1: Unilan's alphabet binary representation

A	E	I	O	U	B	C	D	F	G	J	L	M	N	P	R	S	T	W	Y	Z
00	00	00	00	0	000	000	000	000	000	000	0	000	0 0	000	000	000	000	0 0	0 0	000
0			0	0	0 0 0		0 0 0		0	0 0		0 0 0 0 0 0 0 0 0					0 0 0 0 0			
0	00		0	0	000	0	0 0	000	0 0 0	0 0	0	0 0	000	000	000	000	0 0 0		000	000
0			0	0	0 0 0		0 0 0		0 0 0	0 0	0	0 0 0 0 0			0 0	0	0 0 0		0 0	
0	00	00	00	00	000	000	000	0	000	000	000	0 0 0 0 0			0 0	000	0 000		0	000

And these are some examples of symbols in this alphabet.

FIGURE 2.1: Unilan symbols example (CA - LO - RU)

2.4 Vocabulary

Words are strings of symbols, selected according to its expected frequency of use, and to its semantics (so as to avoid conflicts with words meaning too similar or too opposed meanings).

Each word represents a single concept. There are no variations of a word, no prefixes or suffixes, but since words come on a row, it can be seen as a way of suffixing (adjectives follow nouns, like in Spanish), if read as a single word. In a way, it's like compound words (e.g. 'moonlight').

2.5 Phonetics

Unilan has a phonemic alphabet, that is, each symbol corresponds to a phoneme. The pronunciation is based on the Latin IPA, and sounds similar to the Spanish one, with small differences:

- 'ce' and 'ci' are pronounced like /ke/ and /ki/.
- 'ra', 're', ... 'ru' are always pronounced like /ra/, /re/, ..., /ru/.
- 'ge' and 'gi' are pronounced like /ge/ and /gi/.

Chapter 3

Theoretical study

3.1 Introduction

After we've introduced Unilan, let's move onto the coding part, and let's start talking about what a code is.

Let $\Sigma = \{a_1, ..., a_n\}$ be an alphabet; we call the a_i values **symbols**. A block code C of length n over Σ is a subset of Σ^n. A vector $c \in C$ is called a **codeword**. The number of elements in C, denoted $|C|$, is called the **size** of the code. A code of length n and size M is called an (n, M)-code.

We can represent a code easily in a tree structure. The representation can be done for any size b. The root of the tree is tagged with the empty word, and the nodes are tagged recursively. The children of the node tagged with (w) will be tagged like $w0$ and $w1$ (when $b = 2$).

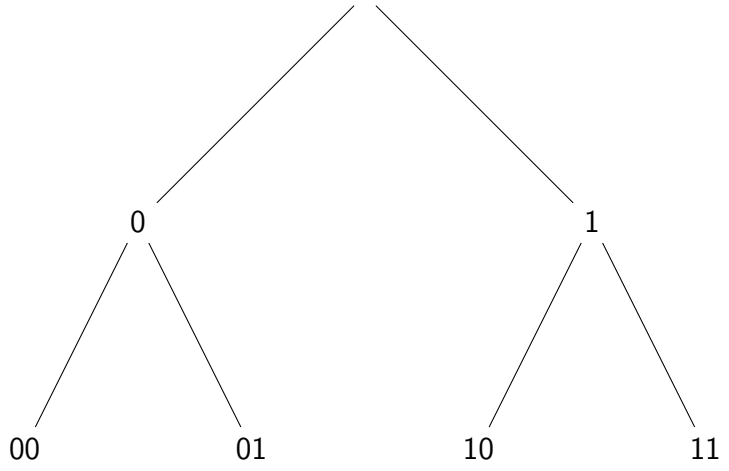
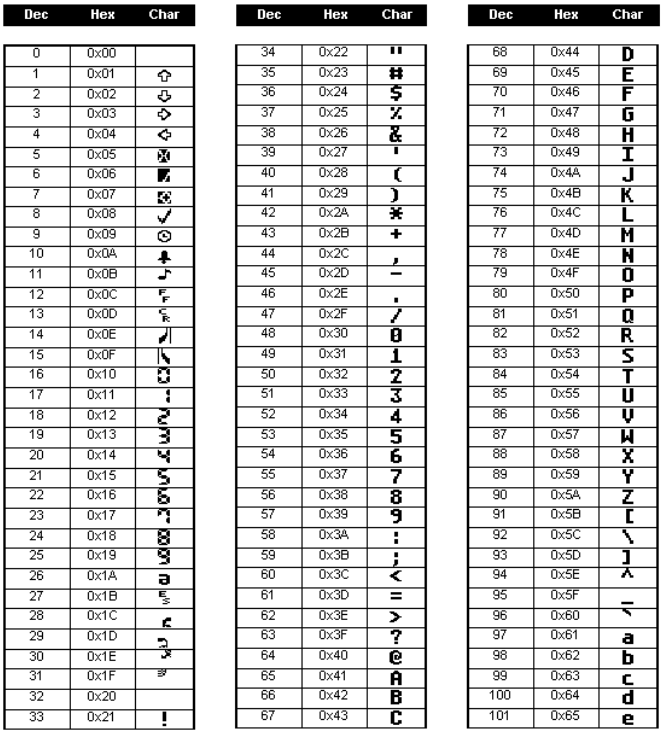


FIGURE 3.1: Binary tree representation

Examples of codes applications are everywhere: Morse, ASCII, QR codes, train tickets...



Dec	Hex	Char
0	0x00	
1	0x01	
2	0x02	
3	0x03	
4	0x04	
5	0x05	
6	0x06	
7	0x07	
8	0x08	
9	0x09	
10	0x0A	
11	0x0B	
12	0x0C	
13	0x0D	
14	0x0E	
15	0x0F	
16	0x10	
17	0x11	
18	0x12	
19	0x13	
20	0x14	
21	0x15	
22	0x16	
23	0x17	
24	0x18	
25	0x19	
26	0x1A	
27	0x1B	
28	0x1C	
29	0x1D	
30	0x1E	
31	0x1F	
32	0x20	
33	0x21	!

Dec	Hex	Char
34	0x22	"
35	0x23	#
36	0x24	$
37	0x25	%
38	0x26	&
39	0x27	'
40	0x28	(
41	0x29	)
42	0x2A	*
43	0x2B	+
44	0x2C	,
45	0x2D	-
46	0x2E	.
47	0x2F	/
48	0x30	0
49	0x31	1
50	0x32	2
51	0x33	3
52	0x34	4
53	0x35	5
54	0x36	6
55	0x37	7
56	0x38	8
57	0x39	9
58	0x3A	:
59	0x3B	;
60	0x3C	<
61	0x3D	=
62	0x3E	>
63	0x3F	?
64	0x40	@
65	0x41	A
66	0x42	B
67	0x43	C

Dec	Hex	Char
68	0x44	D
69	0x45	E
70	0x46	F
71	0x47	G
72	0x48	H
73	0x49	I
74	0x4A	J
75	0x4B	K
76	0x4C	L
77	0x4D	M
78	0x4E	N
79	0x4F	O
80	0x50	P
81	0x51	Q
82	0x52	R
83	0x53	S
84	0x54	T
85	0x55	U
86	0x56	V
87	0x57	W
88	0x58	X
89	0x59	Y
90	0x5A	Z
91	0x5B	[
92	0x5C	\
93	0x5D	]
94	0x5E	^
95	0x5F	_
96	0x60	`
97	0x61	a
98	0x62	b
99	0x63	c
100	0x64	d
101	0x65	e

FIGURE 3.2: ASCII Table

Codes can be divided into two groups, **fixed-length codes** and **variable-length codes**. For example, the ASCII is an example of fixed-length code, because every symbol is coded with the same number of bits.

And what is the advantage of using variable-length codes? Let's think on a book, if there are some letters that appear more frequently, it would be nice if these letters were encoded with a shorter codeword, resulting in a short message, that is, more efficient.

Something like that occurs in a language. Ideally, the more used a word is, the shorter it is. Maybe in the case of Unilan we don't know a priori the expected frequency for each word, but it also would be strange to have a language with all the words with the same length, wouldn't it?

The variable-length codes lead us to the the following topic, called *the decoding problem*:

The symbols appear in a determined order in a message, so we can say that a message is a part of a flow with an established order by a process that happens in real-time.

$$\xi_1\xi_2\xi_3\ldots$$

Each ξ_k is a variable that can take every symbol of the alphabet S as value, and its actual value is the symbol that occurs at instant k $(k = 1, 2, 3, ...)$.

We have a decoding function $c : S \to T^*$ that replaces the symbols in the alphabet S with a coded string from the alphabet T.

Let's consider the alphabet $S = w, x, y, z$ and $c : S \to B^*$ as:
$w \to 10, x \to 01, y \to 11, z \to 011$.

If we receive the string 10011011 we can see that it can represent two different strings:

$$wxwy \to 10011011$$

$$wzz \to 10011011$$

This would be a problem, so it would be reasonable as a requirement that C is a uniquely decodable code, that is, each coded string correspond to a unique original string. Also we would like that the string is decoded in sequential order, without having to wait to the end of the message.

With a prefix coding, we'll ensure that our code is a uniquely decodable code, so next we'll give a definition of what a prefix code is, and what are the techniques for get it.

3.2 Prefix Coding

We'll say that the codeword q is a prefix of another codeword q' if the unique path from q' to the tree root goes through q. A code C will be prefix-free if for all codeword q, any descendant node of q isn't a codeword. This is, we can ignore all the nodes that are descendant of q.

If we ignore all the nodes that aren't codewords nor codeword prefixes, we'll have a finite binary tree, and the codewords of C will be its leafs.

Formally: A *prefix code / prefix-free code / instantaneous code / context-free code* is a type of code system (typically a variable-length code) distinguished by its possession of the "prefix property": there is no code word that is a prefix of any other codeword.

Given the code $L = \{a_1...a_n\}, a_i \in \Sigma^+$ it will be a prefix code (noted $P(L)$), if $\forall \beta \in \Sigma^+, i, j \mid a_i\beta \neq a_j$.

Theorem: If a code $c : S \to T^*$ is prefix-free, then it is uniquely decodable.

On the next part we'll focus on some of the algorithms for creating prefix-free codes.

3.2.1 Algorithms

First, we'll start with the technique introduced by Claude Shannon and Robert Fano[3], the Shannon-Fano coding.

The code is constructed as follows:

- **Shannon-Fano algorithm**

  1. Given a list of symbols, create its probability list or appearing frequency, given as result the relative frequency for each symbol.
  2. Then, sort the symbol lists according to the frequency in descendent order.
  3. After that, the list is divided so that the total frequency of each part are as nearly as possible.
  4. The first part of the list will be assigned the digit '0', and '1' to the second part. That means that the codeword for the symbols in the first part of the list will begin with '0', and the same with '1' for the symbols in the second part of the list.
  5. To finish, apply steps 3 and 4 again to the two sublists, until each sublist corresponds to a symbol.

Here is a implementation of the algorithm in python:

```
def shannon(symbols, count):
    # Step 1
    total = float(sum(count))
    relative_frequencies = []
    for freq in count:
        relative_frequencies.append(freq / total)

    # Step 2
    symbols = sorted(symbols,
                     key=lambda a: relative_frequencies[symbols.index(a)],
                     reverse=True)

    relative_frequencies.sort(reverse=True)

    codewords = [''] * len(symbols)

    shannon_aux(symbols, relative_frequencies,
                codewords, 0, len(symbols) - 1)

    return zip(symbols, codewords)


def shannon_aux(symbols, relative_frequencies, codewords, start, end):
    if start != end:
        # Step 3
        ind_split = 1
        left = relative_frequencies[start]
        right = sum(relative_frequencies[start+1:end+1])
```

```
while ind_split != end:
    next_left = left + relative_frequencies[start+ind_split]
    next_right = right - relative_frequencies[start+ind_split]
    if abs(next_left - next_right) < abs(left - right):
        left = next_left
        right = next_right
        ind_split += 1
    else:
        break

# Step 4
for i in range(start, start + ind_split):
    codewords[i] += '0'
for i in range(start + ind_split, end + 1):
    codewords[i] += '1'

# Step 5
shannon_aux(symbols, relative_frequencies, codewords,
            start, start + ind_split - 1)
shannon_aux(symbols, relative_frequencies, codewords,
            start + ind_split, end)
```

LISTING 3.1: Python implementation of the Shannon-Fano algorithm

And here is the output for next symbol distribution example:

TABLE 3.1: Shannon-Fano example

Symbol	Frecuency	Codeword
A	24	00
B	12	01
C	10	10
D	8	110
E	8	111

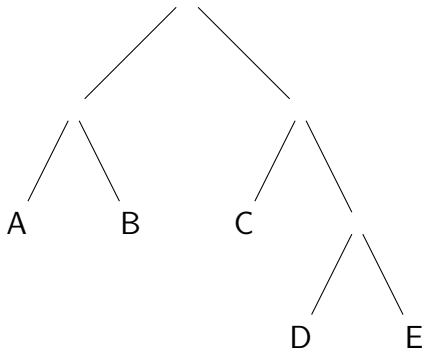


FIGURE 3.3: Shannon-Fano example: Code representation

The previous example results in a code with an average length of 2.26 bits per symbol. However, the Shannon-Fano coding does not always produce the optimal code, and for this reason is

rarely used. Instead of that, the Huffman coding is used, where the average code length is always optimal.

The Huffman coding was introduced in 1952 by David A. Huffman[2]. The algorithm is a typical example of greedy algorithm, whose output always produces an optimal code based on the frequency of each symbol.

Huffman algorithm

1. Create a leaf node for each symbol with the frequency as the node's weight and insert it in the ordered list upwards.
2. While there is more than one node in the list:
    - Remove the two first nodes in the list.
    - Create a new node with a link to the two previous nodes, giving the sum of the weight of both nodes as the new weight.
    - Insert the new node in the list (in the corresponding place according to the weight).
3. The remaining node is the tree root.

And here is the Huffman algorithm implementation in python:

```
from ete2 import Tree, TreeStyle

class CodeTree:
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right

    def save(self, filename):
        tree = Tree('(' + str(self) + ');')
        ts = TreeStyle()
        ts.show_leaf_name = True
        ts.rotation = 90
        ts.show_scale = False
        ts.branch_vertical_margin = 10
        tree.render(filename, tree_style=ts, dpi=300, w=1280)

    def draw(self):
        print Tree('(' + str(self) + ');')

    def __str__(self):
        if self.value is not None:
            return str(self.value)
        left_string = '.'
        right_string = '.'
        if self.left is not None:
            left_string = str(self.left)
            if self.left.value is None:
                left_string = '(' + left_string + ')'
```

```
        if self.right is not None:
            right_string = str(self.right)
            if self.right.value is None:
                right_string = '(' + right_string + ')'

        return right_string + ',' + left_string

class Huffman:
    def __init__(self, distribution_file):
        self.tree = CodeTree(None, None, None)
        self.code_table = {}
        dist = []
        total = 0.0
        for line in open(distribution_file):
            split = line.split('\t')
            total += float(split[1])
            dist.append((CodeTree(split[0], None, None), float(split[1])))
            self.code_table[split[0]] = ''

        # Relative freq
        for index, symbol in enumerate(dist):
            dist[index] = (symbol[0], symbol[1] / total)

        while len(dist) != 1:
            dist.sort(key=lambda pair: pair[1])
            minimum = dist[:2]
            dist = dist[2:]
            new_node = CodeTree(None, minimum[0][0], minimum[1][0])
            dist.append((new_node, minimum[0][1] + minimum[1][1]))

            self.update_table(new_node.left, '0')
            self.update_table(new_node.right, '1')

        self.tree = dist[0][0]
        print self.code_table

    def update_table(self, tree, symbol):
        if tree is None:
            return
        elif tree.value is None:
            self.update_table(tree.left, symbol)
            self.update_table(tree.right, symbol)
        else:
            actual_val = self.code_table[tree.value]
            self.code_table[tree.value] = symbol + actual_val
```

LISTING 3.2: Python implementation of the Huffman algorithm

And for the previous input the result is the following:
In this case the average length is better than before: 2.23 bits per symbol.

TABLE 3.2: Huffman example

Symbol	Frecuency	Codeword
A	24	0
B	12	111
C	10	110
D	8	100
E	8	101

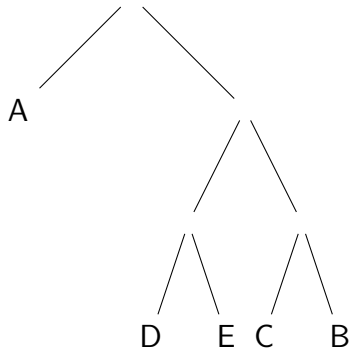


FIGURE 3.4: Huffman example: Code representation

The previous code also can generate the resulting tree, using the python ETE toolkit:

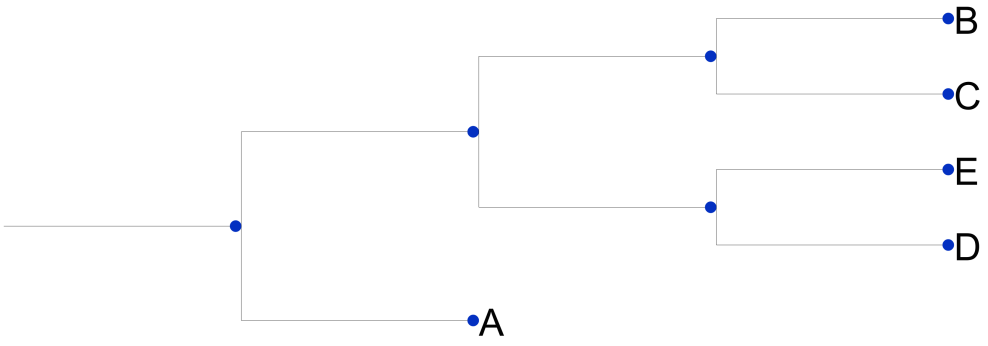


FIGURE 3.5: Tree representation (ETE toolkit)

3.2.2 Extending Huffman

Now that we know how the Huffman algorithm works, we have to think on the constructed language, where there are concepts to encode instead of letters, and there are x symbols instead of '0' and '1'.

So, the first variation to the Huffman algorithm is to extend it for the n-ary support, this variation was also proposed by David A. Huffman his original paper.

- **N-ary Huffman algorithm**

  1. Create a leaf node for each symbol with the frequency as the node's weight and insert it in the ordered list upwards.
  2. While there is more than one node in the list:
     - Remove the n first nodes in the list.
     - Create a new node with a link to the n previous nodes (or i nodes, where i are the latest i nodes remaining, $i < n$), giving the sum of the weight of the nodes as the new weight.
     - Insert the new node in the list (in the corresponding place according to the weight).
  3. The remaining node is the tree root.

Now, there is a little problem with encoding a language with this technique that we've commented before, we don't know a priori the number of words of the language and we don't know the frequency of use of each word, so we need an open word codification, that is, an n-ary tree structure with all the nodes pre-established, where we'll assign a node to a new concept based on its expected frequency of use.

3.2.3 Number of words

In order to determine the number of words necessary for the language we had to refer to the number of words in actual languages dictionaries [4]:

TABLE 3.3: Number of different words for differents languages

Language	Words
Chinese	370.000
English	220.000
Dutch	430.000
French	100.000
German	135.000
Italian	270.000
Japanese	500.000
Korean	500.000
Russian	200.000
Spanish	100.000
Portuguese	390.000

That gives us an idea of the minimum number of words we need. Now, to determine the tree depth and structure, we have done an analysis of the number of words in the Spanish language

by number of syllables, giving us an approximation to the number of nodes that will be on each level of the tree.

The results with the 10.000 more frequent words in Spanish are:

syllables:	1	2	3	4	5	6	7	8
count:	317	2,707	3,896	2,183	679	140	23	6
frequency:	0.03	0.27	0.39	0.22	0.07	0.01	0.002	0.0006

It is remarkable that there are 317 words with only one syllable, far more than we can target in Unilan, since it includes words from 1 to 5 letters (like 'quien'), although some are English words and there are also Roman numerals.

Limiting the number of words of one symbol to 40, and applying the algorithm below to compute the number of different words for each word length, it results in this distribution:

letters:	1	2	3	4
count:	40	1,000	100,000	6,080,000

Another strategy would be to use most monosyllabic words available, with this output:

letters:	1	2	3	4
count:	75	0	10,000	1,760,000

On the other hand, if we limit the number of monosyllabic words to only 12, then we might have a nice set of words made of no more than two letters, being three letters enough to represent a very rich vocabulary (e.g. it might distribute like 12, 0.5k, 300k, 7.6m, what gives room for a huge overall vocabulary. English is estimated to comprise 1m definitions, but most occidental languages handle 100k and 300k, as we have seen, while asiatic ones score higher: 300k to 500k)

letters:	1	2	3	4
count:	12	500	300,000	7,616,000

A smaller alphabet would also work. Say we select only 10 consonants, yielding 50 letters.

letters:	1	2	3	4	
count:	12	1,000	2,000	2,150,000	
letters:	1	2	3	4	5
count:	12	1,000	5,000	1,500,000	25,000,000

Unilan is designed for stability, meaning that words are supposed to stay forever, no matter if the term is used or not. From that point of view, lack of available words could be solved by adding new consonants.

Different words count

Given an alphabet of cardinality s, a prefix code can be defined having x_i words of length i. The number of different words of a given length (once the words of lower lengths have been reserved) is given by this difference equation:

$$n_i = s(n_{i-1} - x_{i-1})$$

where $n_1 = s$

For the sake of clarity, n_i is defined as the maximum number of words for a given alphabet, under a prefix coding, and after reserving $x_1 ... x_{i-1}$ words.

Proof: Given s symbols, we can construct s^i words of length i. In order to meet the prefix condition, we must cancel all the words that derive from reserved words of a lower length. Say that a word of length j (with $j < i$) belongs to the code, then all the combinations that would follow after it, are no longer valid, so s^{i-j} words must be discarded from the s^i pool. Then we can express the number of different words for that length i as

$$n_i = s^i - x_1 s^{i-1} ... - x_{i-1} s^1$$

Adding a coefficient x_0 for the first term with a value that does not alter the equality, we would have this more homogeneous way to express it

$$n_i = -x_0 s^i - x_1 s^{i-1} ... - x_{i-1} s^1$$

where $x_0 = -1$ ($-x_0 = 1$ and $-x_0 s^i = s^i$, so the equation holds).

Studying two consecutive terms in the series

$$n_i = -x_0 s^i - x_1 s^{i-1} ... - x_{i-1} s^1$$

$$n_{i-1} = -x_0 s^{i-1} - x_1 s^{i-2} ... - x_{i-2} s^1$$

we verify that n_i can be rewritten with the terms of n_{i-1} , multiplied by s and adding an extra term at the end

$$n_i = -x_0 s^{i-1} s - x_1 s^{i-2} s ... - x_{i-2} s^1 s - x_{i-1} s$$

and this results in multiplying all the terms by s in this form

$$n_i = s(n_{i-1} - x_{i-1})$$

Implementation: this script in python will compute the maximum number of words for each length, depending on the scheme of distribution of words.

```
def count(s, x):
    # Maximum number of words of a given length for an alphabet and a list of words
    #   s : cardinality of the alphabet, number of symbols (positive integer)
    #   x : number of words of given lengths (list of positive integers)
    #
    #   computes vector n where
    #        n_k = s(n_k-1 - x_k-1)
    #   with n_1 = s
    #   m is the list of maximum values
    #   x is the list of words reserved
    #
    #   example
    #     x = count(80, [12, 1000, 100000])
    #
    #     returns list of length 4, so the next length is computed.

    m = [s]
    n = [s]
    for l in range(1, len(x) + 1):
        m.append(s * m[l-1])
        n.append(s * (n[l-1] - x[l-1]))

    x.append(n[-1])

    return x
```

LISTING 3.3: Python implementation of the number of words algorithm

The final coding will be chosen taking into account different aspects of the language, such as the morphology or semantics.

3.3 Self-Synchronization

3.3.1 Introduction

As future work, we have to look beyond in the use of the language. Even in a scenario of perfect communication (without noise), we all know that no one knows every word from a language (partial knowledge), this implies a problem using the prefix coding because the listener could not know part of the code tree structure that needs to decode the communication flow (does not know a word, for example), so in that moment occurs what is called a synchronization loss, the listener doesn't know where the current word ends and where the next begins.

The codes that allow to recover from this situations are called self-synchronizing codes, although there are several types of self-synchronizing codes.

To speak more precisely about the self-synchronizing properties, we will make some definitions, which can be found in the work of Gilbert and Moore[5]. Given any encoding C and any

1. finite sequences x and y such that x is not the enciphered form (with respect to encoding C) of any message, and xy is a presumed message

if z is a finite sequence of binary digits such that both xyz and yz are complete enciphered messages, we will say that z is a synchronizing sequence for x and y.

Given any uniquely decipherable encoding C, which has some codes of length more than 1, exactly one of the three statements given below will hold:

- For all x and y, there is no z such that z is a synchronizing sequence for x and y. The encoding C will then be said to be never-self-synchronizing.
- For each x and y, there is a z which is a synchronizing sequence for x and y. The encoding C will then be said to be completely self-synchronizing.
- For some x and y there is a synchronizing sequence, but for other u and v there is no synchronizing sequence. The encoding C will then be said to be partially self-synchronizing.

Furthermore, we will define a sequence z to be a universal synchronizing sequence for the encoding C if z is the same synchronizing sequence for all x and y.

Theorem: Given an exhaustive encoding C, then C is completely self-synchronizing if and only if there exists a z which is a universal synchronizing sequence for C.

Now let's consider self-synchronization in Huffman codes[6]:

Let C be a Huffman code. We say C is synchronous if there is a codeword $c = c_1c_2...c_n$ in C satisfying the following two conditions:

1. For all $x = x_1x_2...x_m$ in C such that $m > n$ and c is a substring of x, we have $c_1c_2...c_n = x_{m-n+1}...x_m$ but $c_1c_2...c_n \neq x_ix_{i+1}...x_{i+n-1}$ for any $i \neq m - n + 1$
2. For any $j < n$ such that $c_1c_2...c_j$ can be written as a suffix, the sequence $c_{j+1}c_{j+2}...c_n$ is a string of codewords.

If such a codeword c exists it is called a synchronizing codeword for C. This definition is justified by observing that conditions 1. and 2. guarantee that whenever c is received (without errors), the decoder must automatically resynchronize, regardless of the preceding synchronization slippage.

So, finding a code with a universal synchronizing sequence will be prefered for synchronization loss problems, and the shorter and the more frequent the sequence is, the better.

3.3.2 T-Codes

The original T-Codes were published in 1984 by Titchener[7][8]. This work gave an algorithm for generating families of codes that were self-synchronizing by nature.

This type of codes are **statistically** self-synchronizing, that in best cases can synchronize with about 1.5 codewords of delay.

The T-Code construction algorithm is simple. Code sets are constructed by augmenting lower level T-Code code sets, with the lowest level being the code set 0 and 1 (binary scenario).

The augmentation process consists of writing out a list with two copies of the lower level code set, and then sacrificing a codeword from the first half of the list and using it as a prefix for every codeword in the second half of the list.

This produces a new code set which has nearly twice the number of codewords of the lower level code set. An example of this process is given on the next table.

TABLE 3.4: T-Code example

Level 0	Level 1 Prefix 0	Level 2 Prefix 01
0	→ -	-
1	1	1
	00	00
	01	→ -
		-
		011
		0100
		0101

Because in each iteration of the algorithm the longest codeword increases in at least one symbol, it is not useful for Unilan (at least we want a core language with not so much vocabulary).

3.3.3 Suffix approach

Self-synchronization with suffixes

A idea for making our always-self-synchronizing language, we can think on using suffixes as universal synchronizing sequences.

If our alphabet consists of n symbols and we take a set of symbols S as suffix ($s = |S|$) forcing the language words to end always on some of these symbols ($xy \in C, x \in \Sigma^*, y \in S$), then C will be completely self-synchronizing, that is, if there is a loss of synchronization only we'll have to wait to the current word's end and reach the suffix to recover the synchronization.

This can be seen in the coding tree as follows: Only the symbols from S are leaf nodes, and all the symbols that are in $\Sigma - S$ will be always internal nodes.

The number of words that we'll have using this technique will be (using s symbols from Σ as suffixes):

- s words of length 1
- $(n - s) \cdot s$ words of length 2
- $(n - s)^2 \cdot s$ words of length 3
- ...
- $(n - s)^l \cdot s$ words of length l

For example, for $n = 80$ and $s = 20$:

- 20 words of length 1
- 1.200 words of length 2
- 72.000 words of length 3
- 4.320.000 words of length 4
- ...

Self-synchronization using spaces

Another more natural idea is the use of delimiters (spaces) instead of suffixes. This would result in some advantages:

- A larger word count that with suffixes (same numbers as seen in prefix code section)
- Better legibility

On the other hand, the information flow would be more inefficient (information's overhead).

Chapter 4

Applications

As a way for putting in practice the future language coding, three applications have been developed, a text-to-speech engine (TTS), a first try of a speech recognition system (STT) and a system font for Unilan.

4.1 Text to Speech (TTS)

4.1.1 Introduction

The first 'human speech machines' date back one thousand years, with the "Brazen Head", a legendary automaton that was supposed to be able to answer any question.

But we can say that attempts to synthesize voice have been done since two hundred years ago. For example, in 1791, Wolfgang von Kempelen introduced his "Acoustic-Mechanical Speech Machine", and in about mid 1800's Charles Wheatstone constructed a bit more complicated version of this machine, that was some similar to the figure below.

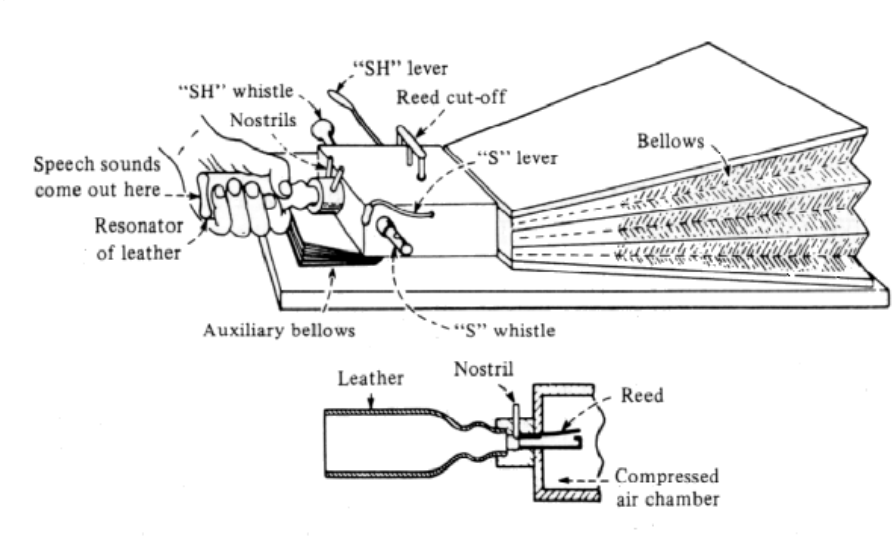


FIGURE 4.1: Wheatstone's reconstruction of von Kempelen's speaking machine.
[9] Taken from Sami Lemmettly.[10]

Over time, the development of the synthesizers moved from mechanical to electrical machines, the first introduced by Stewart in 1922. The machine was able to generate single static vowel sounds with two lowest formants, but not any consonants or connected utterances.

The first device to be considered as a speech synthesizer was VODER (New York World's Fair, 1939), inspired by the VOCODER, developed at Bell Laboratories some years before.

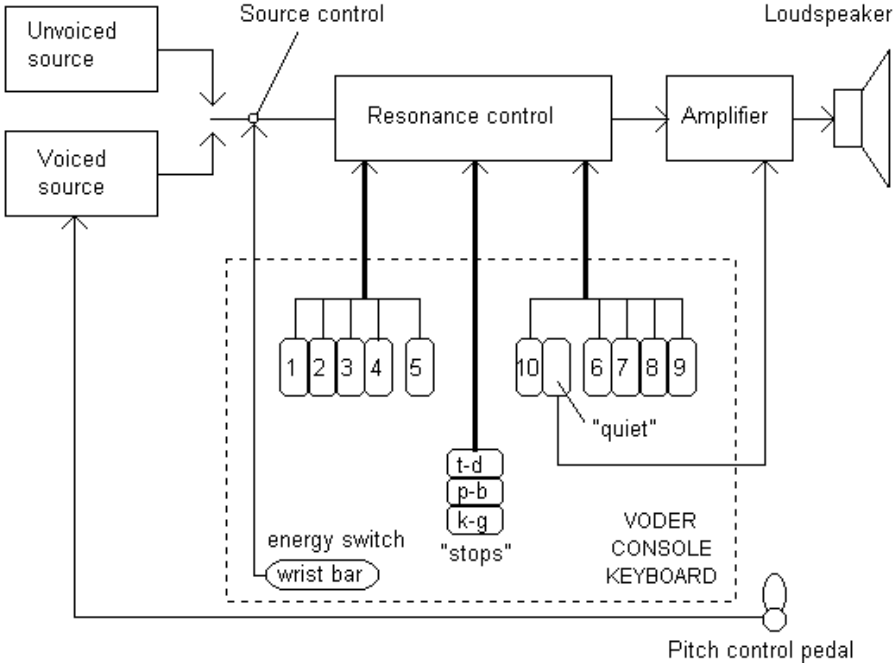


FIGURE 4.2: The VODER speech synthesizer.
[11] Taken from Sami Lemmettly[10].

After the demonstration of the potential of VODER for producing artificial speech, the scientific world became more and more interested in speech synthesis.

Moving along to the text-to-speech systems, their first occurrence were in 1968 in Japan by Noriko Umeda. It was based on an articulatory model and include a syntactic analysis module with sophisticated heuristics.

Since 1970's and early 1980's, a lot of commercial text-to-speech products have been introduced.

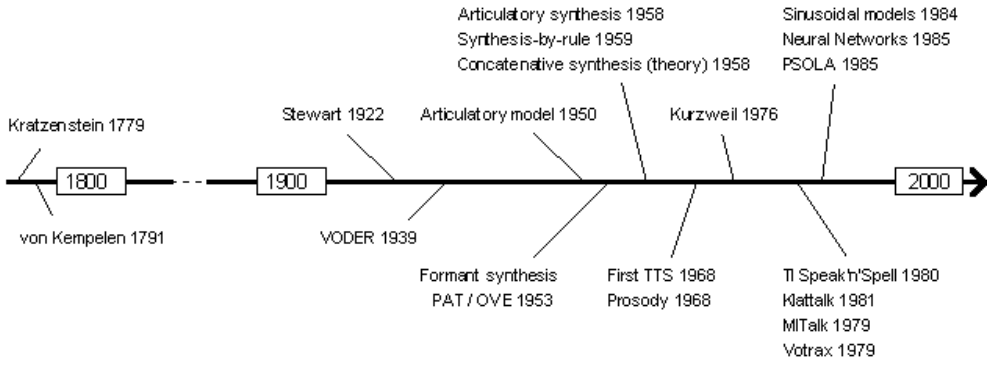


FIGURE 4.3: Some milestones in speech analysis.
Taken from Sami Lemmettly[10].

And finally, we mention here some of the great challenges in the text-to-speech synthesization nowadays:

- Text normalization: Heteronyms (The weather was beginning to **affect** his **affect**), numbers and abbreviations.
- Text-to-phoneme: Text-to-phoneme or grapheme-to-phoneme conversion.
- Prosodics and emotional content.

4.1.2 Current TTS engines

Actual text-to-speech engines are quite advanced, based on hundred on voice recordings, neural networks... some examples of commercial products of tts are:

- Acapela Box
- Google Text-to-Speech
- Cepstral

- ATT natural
- Nuance RealSpeak Voices

However, we'll focus on the open source text-to-speech engines and resources:

- eSpeak[12]: Originally know as speak (1995), uses a "formant synthesis" method. This allows many languages to be provided in a small size. The speech is clear, and can be used at high speeds, but is not as natural or smooth as larger synthesizers which are based on human speech recordings.
- MaryTTS[13]: A multilingual text-to-speech synthesis platform written in Java. Currently supports German, British and American English, French, Italian, Swedish, Russian, Turkish and Telugu (with more languages in preparation).
- Festival[14]: Developed by the the centre for speech technology research at the University of Edinburgh, Festival offers a framework for building speech synthesis systems. Also provides tools for build new voices through Carnegie Mellon's FestVox project.
- MBROLA project[15]: Initiated by the TCTS Lab of the Faculté Polytechnique de Mons (Belgium), its aim is to obtain a set of speech synthesizers for as many languages as possible, and provide them free non-commercial applications.

  Central to the MBROLA project is MBROLA, a speech synthesizer based on the concatenation of diphones. It takes a list of phonemes as input, together with prosodic information (duration of the phonemes and a piecewise linear description of pitch), and produces speech samples (it is therefore not a TTS synthesizer, since it does not accept raw text as input). This synthesizer is provided for free, for non commercial, non military applications only.

4.1.3 Implementation

Due to the language characteristics that Unilan wants to achieve, some of the challenges above mentioned are almost removed, for example:

- Heteronyms doesn't exist, because the desirable lack of ambiguity in the language.
- Text-to-phoneme problem: Totally eliminated, one symbol = one grapheme = one phoneme.
- Prosody is removed, with no intentionality on the pronunciation tone.

First version of the text-to-speech we used eSpeak, with a new definition of sounds for the Unilan language. Then, the software generated the wav files for each of the 80 phonemes of the language. The wav file silences were trimmed and the tts was done using the final wav files with the following python script:

```
import wx
from sys import stdin
from time import sleep


PHONEMES = ['ba', 'be', 'bi', 'bo', 'bu', 'ca', 'ce', 'ci', 'co', 'cu',
            'da', 'de', 'di', 'do', 'du', 'fa', 'fe', 'fi', 'fo', 'fu',
            'ga', 'ge', 'gi', 'go', 'gu', 'ja', 'je', 'ji', 'jo', 'ju',
            'la', 'le', 'li', 'lo', 'lu', 'ma', 'me', 'mi', 'mo', 'mu',
            'na', 'ne', 'ni', 'no', 'nu', 'pa', 'pe', 'pi', 'po', 'pu',
            'ra', 're', 'ri', 'ro', 'ru', 'sa', 'se', 'si', 'so', 'su',
            'ta', 'te', 'ti', 'to', 'tu', 'wa', 'we', 'wi', 'wo', 'wu',
            'ya', 'ye', 'yi', 'yo', 'yu', 'za', 'ze', 'zi', 'zo', 'zu']

WHITESPACE_PAUSE = .3   # Pause (seconds) when a whitespace is read
PHONEME_PAUSE = .05     # Pause (seconds) between phonemes

def main():
    # wx initialization
    app = wx.App(False)
    assert app  # remove pyflakes warning

    sounds = []

    for phoneme in PHONEMES:
        sounds.append(wx.Sound('phonemes/' + phoneme + '.wav'))

    while True:
        fst = stdin.read(1)
        if fst == '\n':
            continue
        if fst == ' ':
            sleep(WHITESPACE_PAUSE)
            continue
        snd = stdin.read(1)
        audio = sounds[PHONEMES.index(fst + snd)]
        audio.Play(wx.SOUND_SYNC)

        sleep(PHONEME_PAUSE)

if __name__ == "__main__":
    main()
```

LISTING 4.1: First text-to-speech script

However, this first version sounded too unnatural, and some phonemes were very difficult to distinguish from others. Because of that, a second version of the phonemes was generated using the MBROLA synthesizer, by only making a phonemic description of each Unilan symbol (e.g. ba = /ba/, ca = /ca/, za = /Ta/, ...). The voice used is also from the MBROLA project (voice es2: Spanish Male (5.1Mb) TCC Communications Corp.).

The phonemes were generated using the following python script, after that, the previous script was used for text-to-speech generation.

```
import subprocess
from tts import PHONEMES

PHONEME_TIME = 150
FILES_PATH = 'mbrola/phon_data/'

OUT_PATH = 'mbrola/out/'

IPA = [['b', 'a'], ['b', 'e'], ['b', 'i'], ['b', 'o'], ['b', 'u'],
           ['k', 'a'], ['k', 'e'], ['k', 'i'], ['k', 'o'], ['k', 'u'],
           ['d', 'a'], ['d', 'e'], ['d', 'i'], ['d', 'o'], ['d', 'u'],
           ['f', 'a'], ['f', 'e'], ['f', 'i'], ['f', 'o'], ['f', 'u'],
           ['g', 'a'], ['g', 'e'], ['g', 'i'], ['g', 'o'], ['g', 'u'],
           ['x', 'a'], ['x', 'e'], ['x', 'i'], ['x', 'o'], ['x', 'u'],
           ['l', 'a'], ['l', 'e'], ['l', 'i'], ['l', 'o'], ['l', 'u'],
           ['m', 'a'], ['m', 'e'], ['m', 'i'], ['m', 'o'], ['m', 'u'],
           ['n', 'a'], ['n', 'e'], ['n', 'i'], ['n', 'o'], ['n', 'u'],
           ['p', 'a'], ['p', 'e'], ['p', 'i'], ['p', 'o'], ['p', 'u'],
           ['r', 'a'], ['r', 'e'], ['r', 'i'], ['r', 'o'], ['r', 'u'],
           ['s', 'a'], ['s', 'e'], ['s', 'i'], ['s', 'o'], ['s', 'u'],
           ['t', 'a'], ['t', 'e'], ['t', 'i'], ['t', 'o'], ['t', 'u'],
           ['w', 'a'], ['w', 'e'], ['w', 'i'], ['w', 'o'], ['w', 'u'],
           ['L', 'a'], ['L', 'e'], ['L', 'i'], ['L', 'o'], ['L', 'u'],
           ['T', 'a'], ['T', 'e'], ['T', 'i'], ['T', 'o'], ['T', 'u']]

def main():
    print 'Generating .pho files...'
    gen_pho_files()

    print 'Generating .wav files...'
    gen_output()

    print 'Done!'

def gen_pho_files():
    for index, phoneme in enumerate(PHONEMES):
        with open(FILES_PATH + phoneme + '.pho', 'w') as pho_file:
            for symbol in IPA[index]:
                pho_file.write(symbol + ' ' + str(PHONEME_TIME) + '\n')

def gen_output():
    for phoneme in PHONEMES:
        subprocess.call(['mbrola', 'mbrola/es1/es1',
                         FILES_PATH + phoneme + '.pho',
                         OUT_PATH + phoneme + '.wav'])

if __name__ == '__main__':
    main()
```

Listing 4.2: Second version of the text-to-speech script

4.1.4 Results

For measuring the clarity of the text-to-speech engine, a little experiment (A total of 33 people, but enough for getting a first impression of the synthesizer) has been done with the objective of identify futures upgrades to the engine, as well as possible confusing sounds (phonemes), that could lead to make some modifications to the Unilan's alphabet, for example.

The experiment consist of: There are a total of 25 words of 2-5 syllables each, and the subjects have to transcribe what they listen to, following the rules of the Unilan language.

We'll call a word success when one person answer correctly to the entire word (all the phonemes of the word are correct). On the next table we can see the success rate for each of the words:

TABLE 4.1: Word success rate results

Word	Success Rate	Word	Success Rate
marimuzu	87.88%	tefapulajo	63.64%
wacusi	84.85%	yorumofeno	60.61%
nelomi	84.85%	jucofutuca	51.52%
siwesulu	75.76%	pifanuno	48.48%
cutadoso	75.76%	dunisazoji	48.48%
denomime	75.76%	fafogiseci	45.45%
farupadana	72.73%	dorewabone	45.45%
powacali	69.70%	gupi	42.42%
binumogari	69.70%	lebugeju	27.27%
patoga	66.67%	wutubili	24.24%
cojuwipewa	66.67%	yudesiwuce	21.21%
bewotoco	66.67%	tizerogoce	9.09%
duyazezu	63.64%		

As well as for the breakdown by phonemes:

TABLE 4.2: Phoneme breakdown success rate results

Phoneme	Success Rate	Phoneme	Success Rate	Phoneme	Success Rate
be	100.00%	me	93.94%	du	84.85%
fa	100.00%	mu	93.94%	yu	84.85%
fo	100.00%	nu	93.94%	zo	84.85%
jo	100.00%	re	93.94%	we	81.82%
le	100.00%	sa	93.94%	wi	81.82%
li	100.00%	ta	93.94%	pa	80.30%
lo	100.00%	ya	93.94%	fe	78.79%
lu	100.00%	zu	93.94%	te	78.79%
ma	100.00%	do	92.42%	to	75.76%
so	100.00%	ga	92.42%	de	72.73%
ju	98.99%	ru	92.42%	ni	72.73%
no	98.99%	ze	91.30%	po	69.70%
ca	98.48%	cu	90.91%	pu	66.67%
fu	96.97%	ne	90.91%	gu	60.61%
la	96.97%	ri	90.91%	tu	60.61%
mi	96.97%	wo	90.91%	bu	57.58%
mo	96.97%	yo	90.91%	pi	57.58%
na	96.97%	bi	87.88%	bo	54.55%
ro	96.97%	ci	87.88%	ge	48.48%
se	96.97%	ji	87.88%	gi	48.48%
si	96.97%	pe	87.88%	wu	46.97%
ce	93.94%	su	87.88%	ti	39.39%
co	93.94%	wa	87.88%	go	27.27%
da	93.94%				

Also, we've analyzed the most likely errors for each phoneme, for example: 'go' is the most confused phoneme, with a success rate of only 27.27% (9/33), 20 of this errors were written as 'wo', so now we know that 'go' and 'wo' are likely to be confused on the synthesizer. Same results for 'ti' (13/33), where all the mistakes were done confusing the phoneme with 'di'.

4.2 Speech to Text (STT)

4.2.1 Introduction

Speech to text, known also as speech recognition (SR), is the translation of spoken words into text.

First investigations about SR are of 1932 in Bell Labs. In 1952 they built a system for single-speaker digit recognition. This system worked using the power spectrum of each sound. Others

milestones in STT includes the IBM's Tangora, that could handle a 20.000 word vocabulary by the mid 1980s or the Sphinx-II system developed by Kai-Fu Lee in 1986. Since then, the technology has moved on and there are plenty of specific techniques for STT.

One of the big problems of speech recognition is facing prosody, as well as inferring the context of the sentence. That's because in natural languages there are a lot of words that sound almost the same, for example, there are phonetic related problems:
"Let us pray"
"Lettuce spray"

Other problems can be syntactic related:
"Meet her at the end of Main Street"
"Meter at the end of Main Street"

Semantic related:
"Is the baby crying"
"Is the bay bee crying"

Or context related:
"It is easy to recognize speech"
"It is easy to wreck a nice beach"

Given the features of the Unilan language, the problems related with the phonetics are practically solved, that is because: the concepts of the language are composed of phonemes, each phoneme is represented by a unique symbol and each phoneme has a unique sound, therefore, we only need a phoneme recognizer for the Unilan speech recognition.

Another of the advantages of the language is because of its encoding, being a prefix free code, spaces are not necessary anymore, and thus neither is it necessary to recognize them.

Next, a basic speech recognition system for Unilan will be proposed, but before that we'll talk a little about the techniques and engines that are used today for speech recognition in natural languages.

4.2.2 Techniques and engines

As we can see in the compilation done in the work by Gaikwad et al.[16], there are a lot of techniques depending on the approach we use. There are four stages for building a speech recognition engine:

1. Analysis
   Done for segmenting the speech signal for further analysis and extracting. Inside this

stage three different techniques are used: Segmentation analysis, sub segmental analysis and supra segmental analysis.

2. Feature extraction
   Several techniques are used in this step for reduce the dimensionality of the input while maintaining the discriminating power of the signal: Principal component analysis (PCA), linear discriminate analysis, independent component analysis...

3. Modeling
   This stage focus on generate speaker models using speaker specific feature vectors. That is, training models with speak data. There are several approaches that can be used in speech recognition process: Acoustic-phonetic, pattern recognition, templated based...

4. Testing
   The system is tested in terms of accuracy, word error rate and speed.

Previous to the realization of the speech recognition algorithm that is proposed here, it was studied if it was possible to apply these techniques to our problem, as in the work done by A. Waibel et al.[17] or the work by Anant G. Veeravalli et al.[18], in which both neural networks and hidden Markov models are used for the construction of a simple phoneme recognizer.

The problem is that for the neural network, as well as for the hidden Markov model, a lot of training data is required for effectively build a reliable system, and we don't have the time to generate that amount of data in this work, so we made a more primitive approach.

Now let's do a quick review to the existing software or tools for speech recognition:

Some of the commercial software more used today includes:

- Dragon Dictation: Developed by Nuance Communications. Based on Dragon NaturallySpeaking technology, is the same that Apple's Siri uses.
- Microsoft Cortana: It uses the natural language processing power from Tellme Networks (bought by Microsoft in 2007).
- Google Voice Search: It supports around 35 different languages, it was created for use of google search by speaking on a mobile phone or computer, now it's integrated along other Google products, like google maps or youtube.

And on the open source side:

- Sphinx[19]: A project by the Carnegie Mellon University. It carries about 20 years of the CMU research. It has support for US English, UK English, French, Mandarin... As

well as tools for build models for other languages. It uses the above mentioned hidden Markov models.

- VoxForge[20]: A set of open source acoustic models for use in speech recognition open source software. All the voices are under the GPL license.
- HTK[21]: The hidden Markov model toolkit developed by the Cambridge University Engineering Department (CUED). As its name indicates, it is a toolkit for build and manipulate hidden Markov models. Using speech data and the transcription of this data, we can generate recognisers that in front of a unknown speech will output its transcription using the HTK training tools.
- Julius[22]: A Japanese large vocabulary continuous speech recognition software based on word N-gram and context-dependent hidden Markov models.

4.2.3 Implementation

As we said before, the solution proposed here is primitive compared to the techniques mentioned above, but it can show us the phonemes that may tend to get confused, or it can serve to be a first speech recognizer system for Unilan, that could be used for a machine-machine effective recognizer (think of robots, for example).

The process for the speech recognizer is the following:

1. First, the audio is splitted into possibles phonemes, for this, a threshold for the space between phonemes is given.
2. The, for each of this chunks:
   (a) The sample is normalized with respect its maximum value.
   (b) The CUDA kernel is called with the sample as an input, that will return the more likely phoneme.
3. All CUDA outputs are concatenated, and that's the recognizer result.

Because of the splitting and trimming of the audio can be imperfect, the CUDA kernel helps with that, using a displacement of +-255 frames of the sample over the phonemes, giving a more accurate match if the some portion of the sample has been lost.

The source code of the implementation is included in Annex A.

4.2.4 Results

Two tests have been designed for the speech recognition system:

1. A first test, to confirm that the system recognizes all the phonemes correctly, when given independently. If the system doesn't pass this test, it's pointless to proceed with the next one.
2. A second test, generating 100 random words of different lengths and confirming that the system recognizes them correctly. This test is designed with the goal of analyzing if the system is capable of correctly identifying the different phonemes in a word, split them into several samples, and identify a whole word right.

For the second test, all the words were generated with a pause between phonemes of 0.03 seconds, and one of the parameters for the AudioMatcher class shown in the annex is the interphoneme separation threshold, that is, how long is a silence for us.

The results we hope to see in this tests are:

That with no silence recognition is that all the phonemes are recognized individually, but any of the words are recognized correctly.

On the other hand, if we set the interphoneme separation threshold to a very high value (in this case more than 0.03 seconds) the results will be the same, because no one word will have enough silence between the phonemes to be identified as two or more phonemes.

The value we're looking for is between 0.00s and 0.03s, so we've performed the test from no silence detection to 0.03s in intervals of 0.001s, and the results for the first test are the following:

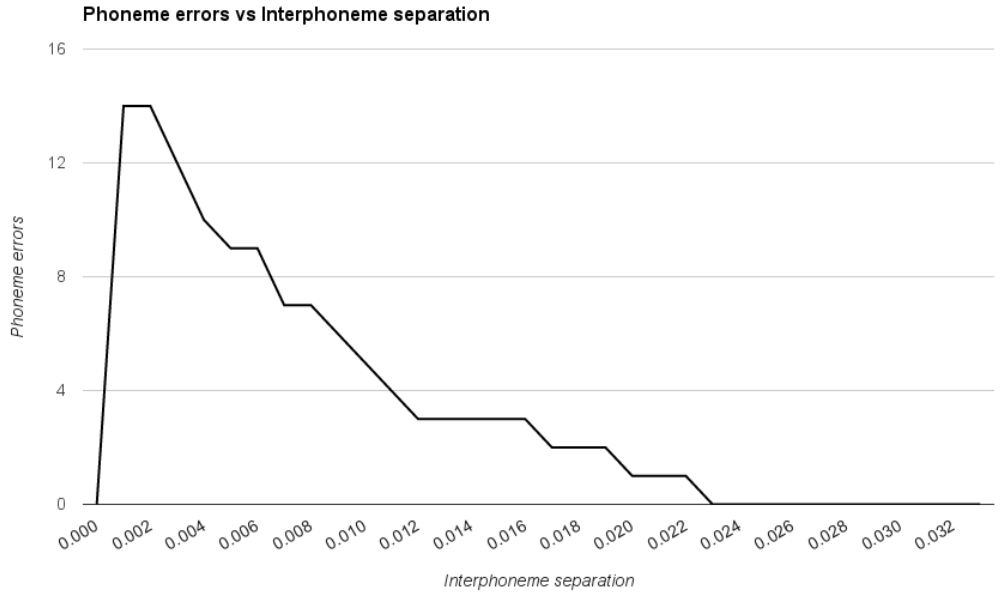


FIGURE 4.4: Phoneme errors vs Interphoneme separation

As we expected, at the beginning there were zero errors. Then, as we start to detect silences, the own phonemes are splitted into two or more samples, so they are incorrectly recognized. From a interphoneme separation of 0.024s, it seems to recognize all the phonemes again.

Now let's move on to the second test.

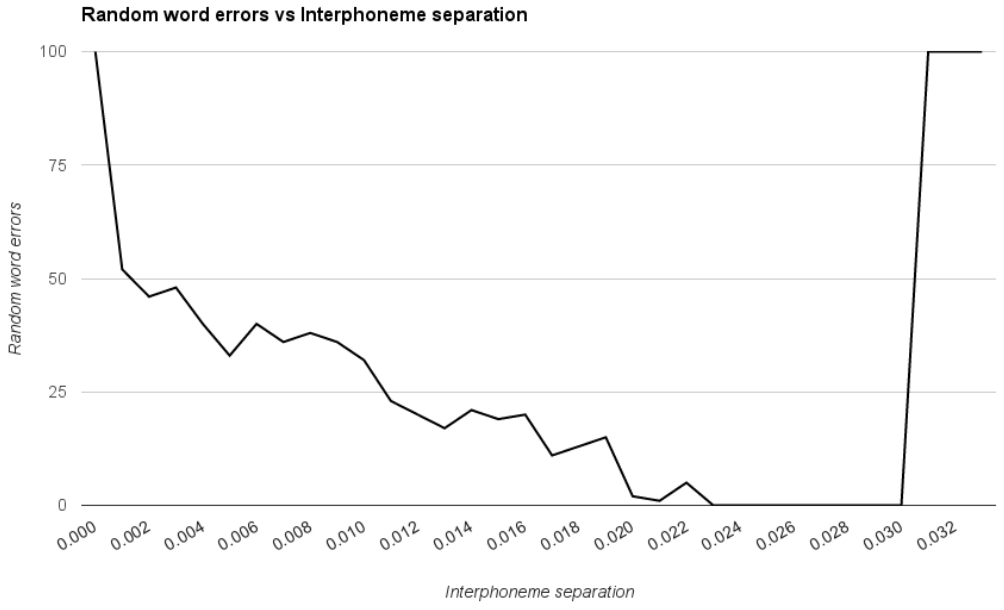


FIGURE 4.5: Random word errors vs Interphoneme separation

Like the previous test, the results are consistent with what we expected. Again, from 0.024s of interphoneme separation all the words are correctly splitted and recognized.

4.3 System Font

The last application we'll talk about is the system font for Unilan. A simple but essential piece for the use of the language on all the desktop operating systems.

The font is generated with the FontForge open source software, and we'll use svg files of the Unilan's symbols to create it.

The script and JSON file for creating the svg files are included in Annex B.

As a result, 21 svg files of the letters are generated, but for some reason, if we try to build the Unilan's alphabet font directly from these files into FontForge, some visualization problems appear after exporting the font file.

The solution is to open all the svg files into some vector editor (Adobe Illustrator, Inkscape) and save them again as new files. For some strange reason, the files we've generated before were causing problems in FontForge, maybe the direction of the strokes are reorganized or the structure of the file is modified somehow after save the file again.

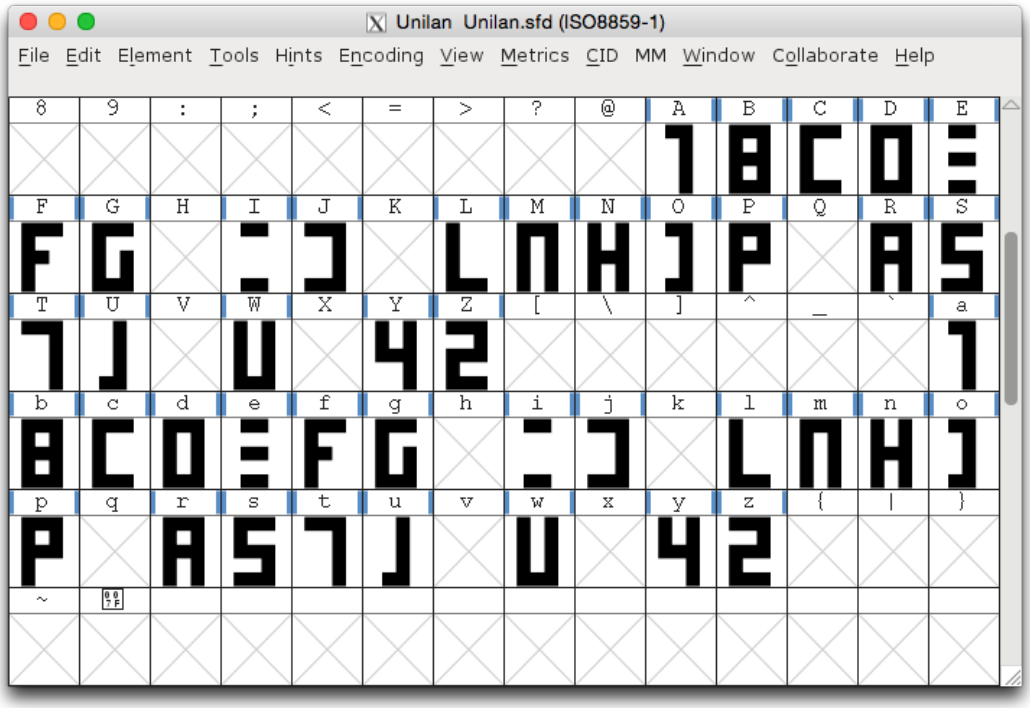


FIGURE 4.6: Building the Unilan's font in FontForge

After that, the Unilan font is ready to be built in FontForge. The way we did it for the Unilan's symbol to appear is to align the vowels to the left part of the available space for the letter, and the consonants to the right side.

A example of the resultant font (the word 'butano') is shown below in a OS X text editor:

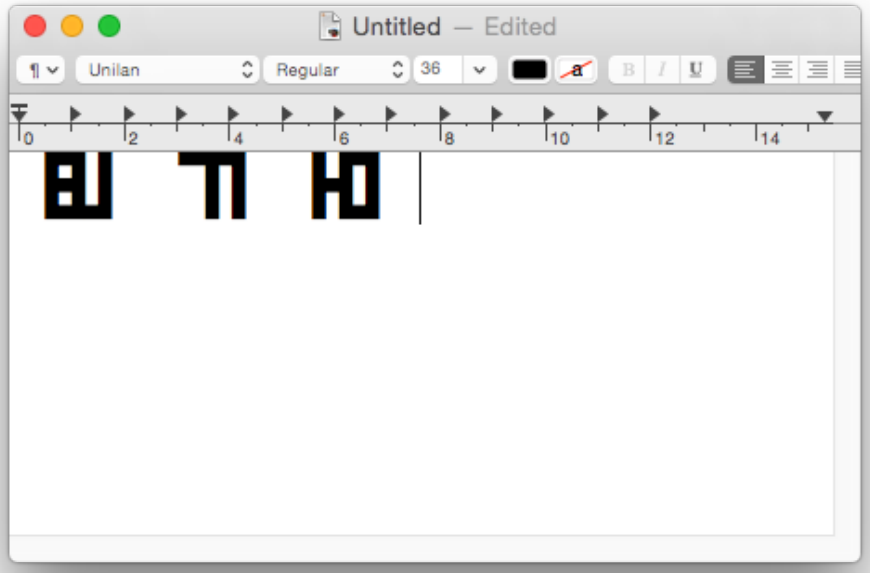


FIGURE 4.7: Unilan's font in text editor

If we consider that the spaces between graphemes are too big, it would be easy to adjust with FontForge in any moment.

To end, an attempt was made to include the Unilan's font in google docs, but it was not possible because its font ecosystem is closed by now. The only option is a google docs plugin that transforms pair of letters to images representing its Unilan's symbol (a pretty useless approach that was discarded after its development).

Chapter 5

Discussion & Conclusions

As we've seen, the work presented covers from theoretical analysis to technical issues.

To begin, we've adapted techniques that were described in the 50s together with linguistic analysis of natural languages that have served us to make a first proposal of a codification for a new constructed language, initially focusing on the prefix codes, for then take a look to the synchronization loss problem, and therefore to the self-synchronizing codes.

To complement the theoretical part, implementations of the different algorithms have been developed in python also.

In the applications part, in addition to the implementations that have been done, the fields of voice synthesization and speech recognition have been studied, obtaining an overview of the current situation on these fields, as well as of some of the problems that the automatic language processing tries to deal with.

In the speech synthesis part we've seen that not always using what Google uses (eSpeak) is going to give us the best result, al least at first. We've chosen for a more simpler approach using existing voices from the MBROLA project, giving us decent results in voice synthesization in a fast way at least.

With the study done we've been able to detect the phonemes that are likely to be confused when using the synthesizer, focusing the future upgrades of the system on these sounds (making the sounds clearer, proposing possible changes to the language design, etc.).

By other side, the fact of doing the study has led to the design and implementation of a web-based plataform for the realization of this via online, so we've dealt with the web development and the use of frameworks for this task also, like for example in this case, Django.

In the speech recognition field, the techniques used today have been studied, watching how the hidden Markov models and the neuronal networks are the most used models in the engines existent nowadays.

As it happened with the synthesis system, we've chosen a simpler approach for the realization of this task, and a development of a system using the techniques mentioned before wouldn't be feasible for the lack of time.

As we commented in its implementation, when detecting and dividing an audio we can lose some of the information of the phoneme we're trying to identify, that is why we have used the CUDA technology that Nvidia provides us for the mass data processing, doing a sort of massive matching with different displacements of the obtained samples over the original phonemes, doing all the comparisons practically without cost, at once.

To end, the development of the system font, that has brought some technical problems with it, like the strange problem of the svg files when trying to use them directly with FontForge, but that had a simple solution, giving as result the last application developed, that still being the most easier to develop, it'll be the most used application for the Unilan language initially.

To finalize and as future work for the language, the following topics are still open:

- The further study of the synthesizer study results, identify the most problematic phonemes and try to fix them. By other side, keep studying other possible solutions using other engines, for example and again, eSpeak.
- The analysis of the possibility of creating an acoustic model of Unilan, for its possible use in some of the studied solutions in the speech recognition part, like sphinx.
- Also contemplate the possibility of improving the developed algorithm, doing a deeper study before of the algorithm (noisy samples, samples spoken by persons, etc.).

Como hemos visto, el trabajo desarrollado aquí cubre desde análisis teórico a la parte más técnica.

Para empezar, hemos adaptado técnicas que fueron descritas en los años 50 en conjunto con análisis sobre lingüística en lenguajes naturales que nos han servido para realizar una primera propuesta de codificación para un nuevo lenguaje artificial, en un principio girando en torno a los códigos libre de prefijo, para en un segundo plano también ver el problema de la pérdida de sincronización, y por ende los códigos auto sincronizantes.

Para complementar la parte teórica, también se han realizado implementaciones de los distintos algoritmos en python.

En la parte de aplicaciones, además de las implementaciones que se han llevado a cabo, se han estudiado los campos de la sintetización y reconocimiento de voz, obteniendo una visión general de la situación actual (el estado de la técnica) en estos campos, así como de algunos de los problemas con los que intenta lidiar el procesamiento automático del lenguaje.

En la sintetización de voz hemos visto por ejemplo que no siempre al utilizar lo que Google utiliza (eSpeak) vamos a obtener un mejor resultado, al menos no de primeras. Hemos optado por un enfoque mucho más fácil utilizando voces ya existentes del proyecto MBROLA, pero que al menos nos ha permitido obtener de manera "rápida" unos resultados decentes en sintetización.

Con el estudio realizado hemos podido detectar los fonemas que más se confunden a la hora de poner a prueba el sintetizador, pudiendo focalizar las posibles mejoras futuras del sistema en esos sonidos (haciendo que estos sonidos suenen más claro, planteando posibles cambios en el diseño del lenguaje, etc.).

Por otra parte, el mero hecho de realizar el estudio ha traído como consecuencia el diseño e implementación de un pequeño sitio web para la realización de éste vía online, por lo que se ha lidiado también con el desarrollo web y la utilización de frameworks destinados para ello, como es en éste caso, Django.

En el reconocimiento de voz, se han vuelto a estudiar las técnicas más usadas actualmente, viendo como los modelos ocultos de Markov y las redes neuronales son los modelos más usuales entre los motores existentes a día de hoy.

Al igual que pasó con el sistema de síntesis, hemos optado por un enfoque más simple para la realización de ésta tarea, ya que un desarrollo de un sistema utilizando las técnicas nombradas anteriormente no sería viable por falta de tiempo principalmente.

Como comentamos en su implementación, a la hora de detectar y dividir un audio podemos perder parte de la información del fonema que intentamos identificar, es por ello que se ha hecho uso de la tecnología CUDA que Nvidia nos provee para el procesamiento masivo de

datos, haciendo una especie de matching masivo con distintos desplazamientos de las muestras obtenidas sobre los fonemas originales, haciendo todas las comparaciones prácticamente sin costo, a la vez.

Por último, el desarrollo de la fuente, que ha traído consigo algunos problemas técnicos, como el extraño problema que daban los ficheros svg al tratar de utilizarlos directamente con FontForge, pero que tenían una solución sencilla al fin y al cabo, dando como resultado la última aplicación desarrollada, que aún siendo la más fácil de desarrollar, será la más utilizada en un principio para el uso del lenguaje Unilan.

Para finalizar y como trabajo futuro para el lenguaje, siguen abiertos los siguientes frentes:

- El estudio en profundidad de los resultados del estudio realizado para el sistema de sintetización, identificar los fonemas más problemáticos e intentar solucionarlos. Por otro lado seguir el estudio de otras posibles soluciones más elaboradas utilizando por ejemplo, de nuevo, eSpeak.
- El análisis de la posibilidad de crear un modelo acústico de Unilan, para su posible uso en algunas de las soluciones estudiadas en la parte de reconocimiento de voz, como sphinx.
- También contemplar la posibilidad de mejorar el algoritmo existente desarrollado, realizando previamente un estudio de campo con más profundidad (muestras con ruido, muestras habladas por personas, etc).

Bibliography

[1] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[2] D. A. Huffman *et al.*, "A method for the construction of minimum redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[3] C. E. Shannon and W. Weaver, "The mathematical theory of information," 1949.

[4] "How many words do I need to know? The 95/5 rule in language learning, Part 2/2." http://www.lingholic.com/how-many-words-do-i-need-to-know-the-955-rule-in-language-learning-part-2/ Accessed: 2015-09-01.

[5] E. N. Gilbert and E. F. Moore, "Variable-length binary encodings," *Bell System Technical Journal*, vol. 38, no. 4, pp. 933–967, 1959.

[6] T. J. Ferguson and J. H. Rabinowitz, "Self-synchronizing huffman codes (corresp.)," *IEEE Transactions on Information Theory*, vol. 30, no. 4, pp. 687–693, 1984.

[7] M. Titchener, "Digital encoding by means of new t-codes to provide improved data synchronisation and message integrity," *IEE Proceedings E (Computers and Digital Techniques)*, vol. 131, no. 4, pp. 151–153, 1984.

[8] M. Titchener, "Generalised t-codes: extended construction algorithm for self-synchronising codes," *IEE Proceedings-Communications*, vol. 143, no. 3, pp. 122–128, 1996.

[9] K. Ishizaka and J. L. Flanagan, "Synthesis of voiced sounds from a two-mass model of the vocal cords," *Bell system technical journal*, vol. 51, no. 6, pp. 1233–1268, 1972.

[10] S. Lemmettly, "Review of Speech Synthesis Technology," 1999.

[11] D. H. Klatt, "Review of text-to-speech conversion for english," *The Journal of the Acoustical Society of America*, vol. 82, no. 3, pp. 737–793, 1987.

[12] "eSpeak: Speech Synthesizer." http://espeak.sourceforge.net Accessed: 2015-09-01.

[13] "The MaryTTS Text-to-Speech System." http://mary.dfki.de Accessed: 2015-09-01.

[14] "The Festival Speech Synthesis System." http://www.cstr.ed.ac.uk/projects/festival/ Accessed: 2015-09-01.

[15] "The MBROLA Project." http://tcts.fpms.ac.be/synthesis/mbrola.html Accessed: 2015-09-01.

[16] S. K. Gaikwad, B. W. Gawali, and P. Yannawar, "A review on speech recognition technique," *International Journal of Computer Applications*, vol. 10, no. 3, pp. 16–24, 2010.

[17] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang, "Phoneme recognition: neural networks vs. hidden markov models vs. hidden markov models," in *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pp. 107–110, IEEE, 1988.

[18] A. G. Veeravalli, W. Pan, R. Adhami, and P. G. Cox, "A tutorial on using hidden markov models for phoneme recognition," in *System Theory, 2005. SSST'05. Proceedings of the Thirty-Seventh Southeastern Symposium on*, pp. 154–157, IEEE, 2005.

[19] "CMU Sphinx." http://cmusphinx.sourceforge.net Accessed: 2015-09-01.

[20] "VoxForge." http://voxforge.org Accessed: 2015-09-01.

[21] "HTK Speech Recognition Toolkit." http://htk.eng.cam.ac.uk Accessed: 2015-09-01.

[22] "Open-Source Large Vocabulary CSR Engine Julius." http://julius.osdn.jp/ Accessed: 2015-09-01.

Appendix A

Speech-to-text (STT) Implementation

The python script for the speech-to-text system (stt.py):

```
import scipy.io.wavfile as wav
import numpy as np
import pycuda.autoinit
import pycuda.driver as drv
import argparse

from pycuda.compiler import SourceModule

assert pycuda.autoinit  # Remove pyflakes warning

CUDA_CODE = SourceModule(open('kernels.cu').read())

# Audio framerate
RATE = 16000

# Silence threshold
SILENCE_THRESHOLD = .001           # <= 0.1% max db

# Minimum space for recognize a silence 'space'
SILENCE_SPACE = int(0.025 * RATE)  # = 0.025 sec

# Minimum length of a recognized phoneme
MIN_LENGTH = int(0.125 * RATE)     # = 0.125  sec

PHONEMES = ['ba', 'be', 'bi', 'bo', 'bu', 'ca', 'ce', 'ci', 'co', 'cu',
```

```
            'da', 'de', 'di', 'do', 'du', 'fa', 'fe', 'fi', 'fo', 'fu',
            'ga', 'ge', 'gi', 'go', 'gu', 'ja', 'je', 'ji', 'jo', 'ju',
            'la', 'le', 'li', 'lo', 'lu', 'ma', 'me', 'mi', 'mo', 'mu',
            'na', 'ne', 'ni', 'no', 'nu', 'pa', 'pe', 'pi', 'po', 'pu',
            'ra', 're', 'ri', 'ro', 'ru', 'sa', 'se', 'si', 'so', 'su',
            'ta', 'te', 'ti', 'to', 'tu', 'wa', 'we', 'wi', 'wo', 'wu',
            'ya', 'ye', 'yi', 'yo', 'yu', 'za', 'ze', 'zi', 'zo', 'zu']


class Audio:
    def __init__(self, filename=None):
        if filename:
            (self.rate, self.data) = wav.read(filename)

    def normalized_data(self):
        new_data = self.data.astype(np.float32)
        new_data = new_data / float(max(new_data.max(),
                                    abs(new_data.min())))
        return new_data

    def save(self, filename):
        """
        Saves the audio in a file

        Arguments:
        filename - Name of the resulting file
        """
        wav.write(filename, self.rate, self.data)

    def trim(self, threshold=SILENCE_THRESHOLD):
        """
        Removes the starting and ending silences from the audio
        """
        normalized = self.normalized_data()
        start_index = 0
        end_index = len(self.data) - 1

        while start_index < len(self.data) \
                and normalized[start_index] <= threshold:
            start_index += 1
```

```
        while end_index > 0 \
                and normalized[end_index] <= threshold:
            end_index -= 1

        if start_index < end_index:
            self.data = np.array(self.data[start_index:end_index],
                                 dtype=np.int16)

    def remove_silences(self, threshold=SILENCE_THRESHOLD,
                        silence_space=SILENCE_SPACE):
        """
        Removes all the silence gaps in the file
        """
        self.trim(threshold=threshold)
        silence_count = 0
        normalized = self.normalized_data()
        new_data = []
        for index, data in enumerate(self.data):
            if abs(normalized[index]) <= threshold:
                silence_count += 1
            else:
                if silence_count >= SILENCE_SPACE:
                    new_data.append(self.data[index])
                else:
                    new_data.extend(self.data[index - silence_count:
                                              index + 1])
                silence_count = 0

        self.data = np.array(new_data, dtype=np.int16)

    def split(self, silence_space=SILENCE_SPACE, threshold=SILENCE_THRESHOLD):
        self.trim(threshold=threshold)
        normalized = self.normalized_data()
        start_index = 0
        end_index = 1
        silence_count = 0

        audios = []
```

```
        while end_index < len(self.data):
            if abs(normalized[end_index]) <= threshold:
                silence_count += 1
                if silence_count == silence_space:
                    silence_count = 0
                    a = Audio()
                    a.rate = self.rate
                    a.data = np.array(self.data[start_index:
                                                end_index - silence_space + 1],
                                      dtype=np.int16)
                    audios.append(a)

                    while end_index < len(self.data) \
                            and abs(normalized[end_index]) <= threshold:
                        end_index += 1
                    start_index = end_index
            else:
                silence_count = 0
            end_index += 1

        if end_index <= len(self.data) and \
                end_index - start_index != silence_count:
            a = Audio()
            a.rate = self.rate
            a.data = np.array(self.data[start_index:
                                        end_index - silence_count + 1],
                              dtype=np.int16)
            audios.append(a)

        return filter(lambda x: len(x.data) > MIN_LENGTH, audios)

    def __str__(self):
        return 'Audio: R=' + str(self.rate) + \
                'Hz L=' + str(len(self.data) / float(self.rate)) + 'S'

    def __repr__(self):
        return self.__str__()


class AudioMatcher:
```

```
    def __init__(self, silence_threshold=SILENCE_THRESHOLD,
                 silence_space=SILENCE_SPACE):
        self.silence_threshold = silence_threshold
        self.silence_space = silence_space

        # Load all the phoneme data
        self.phoneme_data = []
        self.indexes = [0]

        for phoneme in PHONEMES:
            (_, data) = wav.read('phonemes/' + phoneme + '.wav')

            self.phoneme_data.extend(data)
            self.indexes.append(len(data) + self.indexes[-1])

        self.phoneme_data = np.array(self.phoneme_data, dtype=np.uint32)
        self.indexes = np.array(self.indexes, dtype=np.uint32)

    def match(self, audio):
        if audio.rate != RATE:
            raise Exception("The rate isn't " + str(RATE) + "Hz")

        splitted = audio.split(silence_space=self.silence_space,
                               threshold=self.silence_threshold)

        return ''.join([self.match_aux(audio_splitted)
                        for audio_splitted in splitted])

    def match_aux(self, audio):
        total_phonemes = len(self.indexes) - 1
        dest_data = np.zeros((total_phonemes, 512), dtype=np.uint32)
        length = np.array(len(audio.data), dtype=np.uint32)
        source_data = audio.data.astype(np.uint32)

        # Alloc memory in gpu
        source_gpu = drv.mem_alloc(source_data.nbytes)
        dest_gpu = drv.mem_alloc(dest_data.nbytes)
        phonemes_gpu = drv.mem_alloc(self.phoneme_data.nbytes)
        indexes_gpu = drv.mem_alloc(self.indexes.nbytes)
        length_gpu = drv.mem_alloc(length.nbytes)
```

```
        # Copy data to gpu
        drv.memcpy_htod(source_gpu, source_data)
        drv.memcpy_htod(phonemes_gpu, self.phoneme_data)
        drv.memcpy_htod(indexes_gpu, self.indexes)
        drv.memcpy_htod(length_gpu, length)

        # Execute kernel
        grid = (total_phonemes, 1, 1)
        func = CUDA_CODE.get_function("""phoneme_match""")
        func(dest_gpu, source_gpu, phonemes_gpu, indexes_gpu, length_gpu,
             block=(256, 1, 1), grid=grid)

        # Copy result from gpu
        drv.memcpy_dtoh(dest_data, dest_gpu)

        # Free gpu memory
        source_gpu.free()
        dest_gpu.free()
        phonemes_gpu.free()
        indexes_gpu.free()

        result = np.array([min(x) for x in dest_data])

        return PHONEMES[result.argmin()]


def main():
    parser = argparse.ArgumentParser(description='Unilan voice recognizer')
    parser.add_argument('file', metavar='audio_file', nargs=1,
                        help='audio file')

    args = parser.parse_args()

    audio = Audio(args.file[0])
    matcher = AudioMatcher()

    print matcher.match(audio)

if __name__ == '__main__':
```

```
    main()
```

And the CUDA kernel (kernels.cu):

```
__global__ void phoneme_match(int *dest, int *source,
                              int *phonemes, int *indexes,
                              int *size)
{
    int len = size[0];
    int length = indexes[blockIdx.x + 1] - indexes[blockIdx.x];
    int result1 = 0;
    int result2 = 0;
    float compare_index = indexes[blockIdx.x];
    float index_inc = len / (float)length;

    for(int i = 0; i < len - threadIdx.x; i++) {
        result1 += abs(source[i + threadIdx.x] - phonemes[(int)compare_index]);
        result2 += abs(source[i] - phonemes[(int)(compare_index +
                                            threadIdx.x * index_inc)]);
        compare_index += index_inc;
    }

    dest[blockIdx.x * 512 + threadIdx.x * 2] = result1;
    dest[blockIdx.x * 512 + threadIdx.x * 2 + 1] = result2;
}
```

Appendix B

SVG file generation scripts

File UnilanSVG.m:

```
function UnilanSVG (dim)
% unilat.json specifies the unilan version of the latin character map,
% upon which unilan builds.
% each polygon line in a letter is a closed polygon in the width x height grid.
% it goes horizontal first, then vertical, and so on until it closes.
% positive values mean right or down movements, negative mean left or up movements.


folder = './';

u = loadjson ('unilat.json');

if (exist ('dim','var'))
  u.dim = dim;
endif

u.width  = u.width  * u.dim;
u.height = u.height * u.dim;

for idSym = 1:length (u.symbol)
  l = u.symbol{idSym}.letter     % letter
  p = u.symbol{idSym}.polygon;    % shape

  clear pol;
  if (~iscell (p))                % letters with a single polygon are
```

```
    for idDim = 1:size (p, 1)     % read as a vector, not a cell array
      pol{idDim} = p(idDim, :);
    endfor
    p = pol;
  endif

  f = fopen ([folder 1 '.svg'],'wt');     % write svg file
  fprintf (f, '<svg width="%d" height="%d" viewBox="0 0 %d %d">\n',
           u.width, u.height, u.width, u.height);
  fprintf (f, '<path stroke-width="0" d="');
  for idPol = 1:length (p)
    e = p{idPol};
    fprintf (f, 'M%d,%d', e(1) * u.dim, e(2) * u.dim);
    d = 'v';
    for idLin = 3:length (e)
      if (d == 'h')               % alternate directions
        d = 'v';                  % first horizontal, then vertical
      else
        d = 'h';
      endif
      fprintf (f, ' %c%-d', d, e(idLin) * u.dim);
    endfor
    fprintf (f, 'z ');
  endfor
  fprintf (f, '" fill-rule="evenodd" />\n');
  fprintf (f, '</svg>');
  fclose (f);
endfor

endfunction
```

And the unilat.json ('unilat' coming from Unilan-latin characters) file:

```
{
    "dim"   :  5,
    "width" :  3,
    "height":  5,
    "symbol": [
        {
            "letter": "A",
```

```
            "polygon": [
          [ 0, 0, 2, 5,-1,-4,-1]
    ]
      },
      {
          "letter": "B",
          "polygon": [
              [ 0, 0, 3, 5,-3],
              [ 1, 1, 1, 1,-1],
              [ 1, 3, 1, 1,-1]
          ]
      },
      {
          "letter": "C",
          "polygon": [
              [ 0, 0, 3, 1,-2, 3, 2, 1,-3]
          ]
      },
      {
          "letter": "D",
          "polygon": [
              [ 0, 0, 3, 5,-3],
              [ 1, 1, 1, 3,-1]
          ]
      },
      {
          "letter": "E",
          "polygon": [
              [ 0, 0, 2, 1,-2],
              [ 0, 2, 2, 1,-2],
              [ 0, 4, 2, 1,-2]
          ]
      },
      {
          "letter": "F",
          "polygon": [
              [ 0, 0, 3, 1,-2, 1, 1, 1,-1, 2,-1]
          ]
      },
      {
```

```
        "letter": "G",
        "polygon": [
            [ 0, 0, 3, 1,-2, 3, 1,-2, 1, 3,-3]
        ]
    },
    {
        "letter": "I",
        "polygon": [
            [ 0, 0, 2, 1,-2],
            [ 0, 4, 2, 1,-2]
        ]
    },
    {
        "letter": "J",
        "polygon": [
            [ 0, 0, 3, 5,-3,-1, 2,-3,-2]
        ]
    },
    {
        "letter": "L",
        "polygon": [
            [ 0, 0, 1, 4, 2, 1,-3]
        ]
    },
    {
        "letter": "M",
        "polygon": [
            [ 0, 0, 3, 5,-1,-4,-1, 4,-1]
        ]
    },
    {
        "letter": "N",
        "polygon": [
            [ 0, 0, 1, 2, 1,-2, 1, 5,-1,-2,-1, 2,-1]
        ]
    },
    {
        "letter": "O",
        "polygon": [
            [ 0, 0, 2, 5,-2,-1, 1,-3,-1]
```

```
        ]
    },
    {
        "letter": "P",
        "polygon": [
            [ 0, 0, 3, 3,-2, 2,-1],
            [ 1, 1, 1, 1,-1]
        ]
    },
    {
        "letter": "R",
        "polygon": [
            [ 0, 0, 3, 5,-1,-2,-1, 2,-1],
            [ 1, 1, 1, 1,-1]
        ]
    },
    {
        "letter": "S",
        "polygon": [
            [ 0, 0, 3, 1,-2, 1, 2, 3,-3,-1, 2,-1,-2]
        ]
    },
    {
        "letter": "T",
        "polygon": [
            [ 0, 0, 3, 5,-1,-4,-2]
        ]
    },
    {
        "letter": "U",
        "polygon": [
            [ 0, 4, 1,-4, 1, 5,-2]
        ]
    },
    {
        "letter": "W",
        "polygon": [
            [ 0, 0, 1, 4, 1,-4, 1, 5,-3]
        ]
    },
```

```
        {
            "letter": "Y",
            "polygon": [
                [ 0, 0, 1, 2, 1,-2, 1, 5,-1,-2,-2]
            ]
        },
        {
            "letter": "Z",
            "polygon": [
                [ 0, 0, 3, 3,-2, 1, 2, 1,-3,-3, 2,-1,-2]
            ]
        }
    ]
}
```