A Genetic Algorithm Approach for Verification of the Syllable Based Text Compression Technique

Göktürk Üçoluk İ. Hakkı Toroslu

Department of Computer Engineering
Middle East Technical University

Emails: ucoluk@ceng.metu.edu.tr
toroslu@ceng.metu.edu.tr

Abstract

Provided that an easy mechanism exists for it, it is possible to decompose a text into strings that have lengths greater than one and occur frequently. Having in one hand the set of frequently occurring such strings and in the other the set of letters and symbols it is possible to compress the text using Huffman coding over an alphabet which is a subset of the union of these two sets. Observations reveal that in most cases the maximal inclusion of the strings leads to an optimal length of compressed text. However the verification of this prediction requires the consideration of all subsets in order to find the one that leads to the best compression. A Genetic Algorithm (GA) is devised and used for this search process. Turkish texts, where due to its aglunitative nature, the language provides a highly regular syllable formation, are used as a testbad.

1 Introduction

1.1 Text Compression and Huffman Coding

Though the cost of computer storage devices has dropped dramatically and far more storage is available to the user it can still be argued that the need for data compression is preserving its significance since the information nowadays being stored on electronic media is also exponentially growing. Data compression is used in practice for two purposes, namely *data storage* and *data transmission.* By making use of various compression techniques typically storage savings of 20% to 50% for text files might be achieved. Due to large homogeneous patterns these figures escalate to 50% to %90 for binary files [9]. Two main techniques are made use in data compression: *run-length encoding* and *variable-length encoding* [4]. Run-length encoding is simple and is based on storing the numbers of successively repeating patterns. Therefore it is not very suitable for text compression since the only repeating character that is likely to be found in a text is the blank character. Variable-length encoding is making use of the frequency information of the patterns. After a statistical analysis of the data, frequently occuring patterns are assigned shorter codes than the ones that occur less frequently. D. Huffman discovered a general method for finding the optimal coding of the data using the frequencies of the patterns [9]. In this way a binary tree which has at its leafs the patterns is constructed *(such a tree is called a 'trie').* The way to reach a leaf is to start from the root and branch left or right on each node on the path. This sequence of branching information (lets say a 1 for a left and a 0 for a right) is the *encoding* of that specific leaf. In a Huffman trie patterns at the leafs that are placed closer to the root are the ones that frequently occur in the data. Hence reaching them requires lesser branchings which results in shorter codes for them. Here is an example:

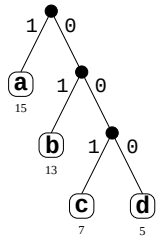
Assume we are encoding the text

```
abababababababccbbbbbbbddaaaaaacdcdcdacac
```

The letter frequencies are as follows:

Letter	*Frequency*
a	15
b	13
c	7
d	5

Since our alphabet a,b,c,d has four elements for uncompressed denotation each letter would require 2 bits ($\lg 4 = 2$)[1] *(Uncompressed denotation is different than the ASCII representation which requires 8 bits/char. Commiting to the ASCII representation implicitely means that there are 256 distinct characters. In section 5 this is explained in more details).* The text contained 40 characters. So, in total, $2 \times 40 = 80$ bits would be required. The Huffman tree is constructed as



So the coding (at bits) level is as follows:

Letter	*Code*	*Code length*
a	1	(1 bit)
b	01	(2 bits)
c	001	(3 bits)
d	000	(3 bits)

For the above given text the sum of required bits can be calculated as follows:

$$\underbrace{15 \times 1}_{\texttt{a}} + \underbrace{13 \times 2}_{\texttt{b}} + \underbrace{7 \times 3}_{\texttt{c}} + \underbrace{5 \times 3}_{\texttt{d}} \;=\; 77 \text{ (bits)}$$

So a compression ratio of $77/80 = 0.9625$ is obtained.

[1] lg represents $\log_2$

1.2 Making use of the repeating patterns in the text: Extending the alphabet

Again consider the above given text example, but this time we indicate the repeating patterns for sake of easy recognition *(superscript stands for repetition, parenthesis for grouping)*:

$$(\mathtt{ab})^7\mathtt{c}^2\mathtt{b}^6\mathtt{d}^2\mathtt{a}^6(\mathtt{cd})^3(\mathtt{ac})^2$$

Without making use of any additional ordering information in the text other than the grouped symbols there are various possible alphabets in terms of which we can express the above text. These are

$$\begin{array}{ll} \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\} & \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab}\} \\ \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ac}\} & \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{cd}\} \\ \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab},\mathtt{ac}\} & \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab},\mathtt{cd}\} \\ \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ac},\mathtt{cd}\} & \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab},\mathtt{ac},\mathtt{cd}\} \end{array}$$

Constructing the corresponding Huffman trees one obtains

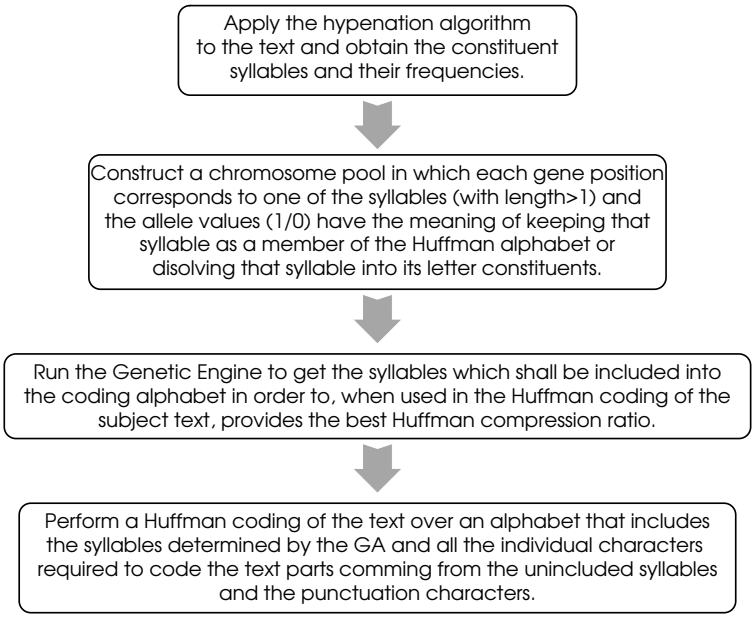
Alphabet	Uncompr. Len.	Compr. Len.	Compr. Ratio
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$	80.0	77	0.9625
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab}\}$	76.6	77	1.0049
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ac}\}$	88.2	82	0.9294
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{cd}\}$	85.9	73	0.8497
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab},\mathtt{ac}\}$	80.1	80	0.9983
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab},\mathtt{cd}\}$	77.5	74	0.9543
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ac},\mathtt{cd}\}$	90.5	75	0.8290
$\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{ab},\mathtt{ac},\mathtt{cd}\}$	78.6	74	0.9414

This small example proves that it is possible to perform better than only letter based Huffman coding. Furthermore there is no theoretical easy mechanism which pinpoints the alphabet that will lead to the best compression. Experience with lengthy texts shows that the subset of the alphabet set which constitutes of single letter remains *almost* constant and at most looses 1-2%. But no general rule has

been identified for the strings with lengths greater than one. This results in a search process over all members of the power set of the set of such strings (with lengths greater than one). Even with very regular languages that have relatively small number of syllables that repeat, the order of the set size is of magnitude 10^3. As well known, the enumeration of all the members of a power set of a set with N members is an $\mathcal{O}(2^N)$ process. Hence, a productive search method is needed to approach the optimal.

A Genetic Algorithms (GA) technique is employed for this purpose. GA require a mechanism for evaluation of the success of candidate solutions which is in our case the Huffman compression lengths obtained by using the candidate alphabets. But this requires extensive Huffman tree constructions each of which is known to be of order $\mathcal{O}(N \lg N)$ with a considerably large constant time factor due to its complicated data structure. So, a theoretical approximation for the compressed text length is used which enables to estimate the compressed length using only mathematical operations on the repetition counts of the alphabet members in the text. The computational complexity of this approximation is of order $\mathcal{O}(N)$ where the constant factor can considerably be reduced by pre-calculation.

The proposed method is tested with Turkish corpora which possesses a regular syllable structure. Due to this regularity of the language the count of used syllables in a huge sized corpora remains of the order of a few thousands. It is observed that about two hundred of them occur with a high frequency. Furthermore Turkish language provides a very grammatical and simple algorithm for hyphenation which produces the syllable boundaries of any word. The following flowchart summarizes the whole compression process.



The next section explains this approximation technique and in the section 3 the GA is presented. Section 4 briefly summarizes the syllable formation of Turkish and the algorithm used to obtain the syllables. In 5, the last section, we present the results/observations hence obtained and conclude.

2 Theoretical Approximation for the Compressed Text Length

Based on Shannon's contribution, it can be proven [4, 8] in Coding Theory that

> *If the entropy of a given text is $\mathcal{H}$, then the greatest lower bound of the compression coefficient μ for all possible codes is $\mathcal{H}/\lg m$ where m is the number of different symbols of the text.*

On the other hand we know that, modulo coding alphabet, Huffman coding is optimal. So, we can conclude that

$$\mu = -\frac{1}{\lg m} \sum_{i=1}^{m} p_i \lg p_i \tag{1}$$

is a good approximation for Huffman compression. Here p_i is the probability of a symbol, member of the alphabet, to occur in the text and is defined as n_i/n where n_i is the count of the ith alphabet member in the text and n is the total count of symbols in the text ($n = \sum_{i=1}^{m} n_i$). The real Huffman compression coefficient will be equal to or slightly greater than this theoretical upper bound.

If we are interested in the final code length which is the actual measure for the usefulness of any data compression, this compression coefficient must be multiplied by the bit-length of the uncompressed text in order to obtain the bit-length of the compressed text. The bit-length of the uncompressed text is $n \lg m$. Hence, given an alphabet, the theoretical lower bound for the bit-length of the compressed text is:

$$l_{compressed} = n \lg n - \sum_{i=1}^{m} n_i \lg n_i \quad \text{[bits]}. \tag{2}$$

Note that this quantity is *not* invariant under alphabet changes since n, m and n_i values will vary from alphabet to alphabet.

3 GA Representation

Genetic Algorithms are employed for algorithmic searches by mimicking the way nature genetically searches for solutions of the survival problem of its species. In GA a population of individual objects, each possessing an associated fitness value, are transformed into a new generation of the population using Darwinian principle of reproduction. Here using the genetic operations such as recombination (crossover) and mutation the population is breaded and evaluated with a 'survival of the fittests' criteria. Each individual of the population represents a possible solution to the given problem. The GA attempts to find the optimal or nearly optimal solutions to the problem by applying those genetic operations to the population of individuals. In practice GA is astonishingly efficient in searching complex, highly non linear, multidimensional search spaces. Excellent reviews can be found in [2, 1, 10].

The main ingredient of a GA implementation is the *chromosome* which is the encoding of the parameters of a possible solution of the subject problem. This encoding can be done in various ways but mostly a fixed length string is chosen due to the relative easiness of the implementation of genetic operations. Each parameter corresponds to a fixed position in the chromosome which is named as a *gene*. The set of possible valid values a gene can hold is called the *allele* of that gene. A function which evaluates the success of a solution, namely a chromosome, is devised. This function is named as the *fitness function* and maps the space of chromosomes to $\mathcal{R}$. The *run* of a GA is a two step process. First an initialization of a pool of solutions is performed. This creation of the initial pool is mostly done randomly. Then the main step of the GA is entered. This is a controlled cycle of a process which constitutes of three basic steps. Each iteration of this cycle is called a *generation*. In each generation the population of chromosomes are *mutated* (randomly genes are replaced by other allele members) and then mated for *crossover* in which chromosomes exchange genes with their partners in a random manner with predetermined probabilities. Following this, an evaluation process is carried out over the new generated population. Using the values obtained from the fitness function a decision is made about the chromosomes whether to exist (to live) in the next generation or not (to die).

In our problem to every predetermined possible constituent string of the text (namely syllable) with length greater than one a *gene* is assumed to correspond. A *chromosome* is a fixed length vector of *allele* values `1` or `0`. A chromosome corresponds to a candidate solution (in our case a candidate alphabet). A gene value of `1` means the corresponding string is included into the alphabet. A `0` means the corresponding string will be *dissolved* into its letters resulting in an incremental contribution to the count of those letters. A fitness function, devised, evaluates a chromosome by first dissolving the not included strings, and then calculating the compressed length which would be obtained by using that alphabet in the Huffman coding. This calculation uses the theoretical approximation explained in the previous section. The compression length serves as the *fitness value* which will be used to determine which chromosome is going to live and which is going to

die. Since the optimal compression length is not known the fitness value cannot be converted to an absolute fitness criteria, but rather be used as an *ordering relation*. Hence it is used to obtain a sorting among the solutions. After a new generation is produced it is sorted according to each chromosome's fitness value. The bests of the previous generation replace the worsts of the new generation provided that

- This replacement takes place for a predetermined percentage of the pool at most *(Which is about 5%-10%)*.
- The fitness of the replacing chromosome is better than the replaced one.

The GA engine has the following algorithmic outline. *In order to give an idea about the tuning of the engine the dynamic settings used in the real application in which about 2700 genes/chromosome existed are also mentioned in italics.*

- Generate a random population *(see explanation below)*, evaluate it and store it also as the former generation.
  Pool size = 100 chromosomes

Repeat :

- Mutate.
  Mutation Rate = Once each 10 Generation one random chromosome
  Changes per Chromosome = Flip one randomly selected gene
- Mate all the pool forming random pairs, then perform random crossovers among pairs. Hence form a new generation.
  Cross Over = At 10 random chosen random length gene intervals
- Evaluate the new generation, using the theoretical approximation. Replace the pool by the new generation keeping the 'real bests' of the previous generation. *Keep Ratio = At most 10% (see above text for replacement condition)*
- Display/Record performance result.
- If it was not the last generation the user demanded, **goto Repeat**.

Although GA, by nature, is a mechanism in which the optimal solution is approached eventually, the speed of convergence heavily depends on some aspects of the GA process. The initial pool creation is one of these aspects. If the global properties of the specific problem does not provides clues about the region the search space has to be searched for the optimal, then it is wise to include members from almost all regions in the initial population. In the subject problem, the initial population had to be created from the elements of the power set of all constituents of the text. The evenness of the selection is obtained by including equal number of members of almost all possible cardinalities. So, for example, the pool certainly contained a member which corresponded to the inclusion of all the constituents, another which corresponded to the exclusion of all constituents except the letters and others which have inclusion ratios linearly distributed between these two extremes. The pool size of 100 is a good value accepted empirically by many GA researchers [3]. The use of the crossover operator is encouraged for GA problems with encoding schemes that promotes the propagation and formation of useful 'building blocks' under this operation. The encoding used in this application is of such a nature. The crossover used is a restricted version of uniform crossover. As known, in uniform crossover each gene is crossed with a certain probability *(for a good review of the subject see [6, 5])*. We restrict the probability to 0.5 and the total count of swapped blocks of genes to 10. Various experimentations with these values proved that 10 is an appropriate (though not vary critical) choice for proper convergence. The selection phase uses an elitist approach where at most 10 elements of the previous generation are kept 'alive' provided that they are better than the worst elements of the current generation. It has been observed that such an approach stabilizes the convergence of the GA. The number of iteration to converge to a solution is about 200-400 generations.

4 Turkish Syllable Formation

Turkish is an aglunitative language. Inflections, tenses are all generated by appending several suffixes to root words. Although not very common, it is possible

to observe meaningful word formations which make use of 10-15 suffixes. The syllable boundaries in a word are the hyphenation points and the language has a simple algorithm for hyphenation. Turkish language has 8 vowels and 21 consonants. Any syllable of the language has to have exactly one vowel. In a syllable at most two adjacent consonants is allowed. Some additional rules limit the possibilities to the followings *(*$\boxed{\mathbf{V}}$*:Vowel,* $\boxed{\mathbf{C}}$*:Consonant)*:

Syllable Pattern	*Number of Possibilities*
$\boxed{\mathbf{V}}$	8
$\boxed{\mathbf{C}}\boxed{\mathbf{V}}$, $\boxed{\mathbf{V}}\boxed{\mathbf{C}}$	336
$\boxed{\mathbf{V}}\boxed{\mathbf{C}}\boxed{\mathbf{C}}$, $\boxed{\mathbf{C}}\boxed{\mathbf{V}}\boxed{\mathbf{C}}$	7056
$\boxed{\mathbf{C}}\boxed{\mathbf{V}}\boxed{\mathbf{C}}\boxed{\mathbf{C}}$	74088

Though this combinatorial calculation reveals 8×10^4 possibilities, the language uses about 3% of them only. The most frequent types of occurances are the of $\boxed{\mathbf{C}}\boxed{\mathbf{V}}$, $\boxed{\mathbf{V}}\boxed{\mathbf{C}}$, $\boxed{\mathbf{C}}\boxed{\mathbf{V}}\boxed{\mathbf{C}}$ type patterns. The four letter combinations are extremely rare (of order 10^2) and occur in a text with probabilities of order 10^{-4}.

The following simple algorithm which is $\mathcal{O}(n)$*,(denoted in C syntax)*, linearly scans a word and produces the syllables.

Assume the character array `turkish_word[]` holds the word to be hyphenated and the function `hyphen()` upon a call fills out an array of pointers `syl[]` that has elements pointing to the *end*s of successive syllables. Furthermore A global integer variable `syl_count` gets set by `hyphen()` to the count of syllables formed.

```
char *twp;

void hyphen()
{
  syl_count=0;
  twp=turkish_word-1;
  do
     { if (next_is_vowel())
         if (next_is_vowel()) mark(1);
         else loop: if (next_is_vowel()) mark(2);
                    else if (*twp) goto loop; }
  while (*twp);
  while (!*--twp);
  mark(0);
}

void mark(char k)
{ syl[syl_count++] = (twp-=k)+1; }

int next_is_vowel()
{ if (is_vowel(*(++twp)) return 1 else return 0; }

int is_vowel(char c)
```

returns `1` *if* `c` *is a vowel else returns* `0`

The existence of such a simple syllable formation algorithm and the relatively small number of distinct syllables that are used frequently throughout the language enables an efficient use of the proposed compression technique for Turkish texts.

5 Results for Turkish Texts and Conclusion

A total of 5 MBytes of Turkish corpora, mainly gathered from news items and magazine articles, are compressed through this algorithm and compared size wise with the standard adaptive Huffman coding over a letter alphabet. The found results are tabulated below.

		Corpus$_1$	**Corpus$_2$**[2]	**Corpus$_3$**	**Corpus$_4$**
Original ASCII text file size *(8 bits/char)*	# of bytes	203796	815927	1103054	2658982
	# of bits	1630368	6527416	8824432	21271856
Length of uncompressed representation with min. # of bits/char *(see text)*		1222776	4079635	6618324	15953892
Huffman coding over single character alphabet	Length *(bits)*	1054033	3473728	5479973	12922652
	Compression ratio (μ) w.r.t. minimal bit representation	0.86	0.85	0.83	0.81
	Compression ratio w.r.t. ASCII size	0.65	0.53	0.62	0.61
Huffman coding over GA determined syllable extended alphabet	Length *(bits)*	796052	2858622	4447514	10848646
	Compression ratio (μ) w.r.t. minimal bit representation	0.65	0.70	0.67	0.68
	Compression ratio w.r.t. ASCII size	0.49	0.44	0.50	0.51
	(# of kept syllables)/ (Total # of syllables)	1976/ 1983	2371/ 2371	2171/ 2204	2579/ 2594

Compared to the standard coding the method has provided an up to 21% better compression ratio (excluding the overhead which is negligible). The worst improvement observed was 13%. It is worth to point out the difference of this quantity with the usually (but mistakenly) used quantity. Usually a compression reference is made by putting down the value of reduction in the file size. Text files are made of bytes where each character (letter, punctuation symbol, etc.) is represented by a unique byte (usually the ASCII code). But the whole bunch of

[2]contains only lower case letters, hence can be coded with 5 bits.

the 'used' characters are much less then 2^8 so actually a considerable number of byte patterns have no character association at all or are not found in the text file. Hence it is wrong to assume that the original alphabet of the compression consists of 2^8 distinct characters (this is assuming that the uncompressed data has a greater information content than it actually possesses). Therefore the compression ratio (which we refer to as μ) is properly calculated with respect to the real information size of the uncompressed text. This means if the original text consists of L characters from a set of N distinct characters, since $\lceil \lg N \rceil$ bits would be sufficient to represent each character, the whole text would require $L \times \lg N$ bits. After compression, assume that the compression yields C number of bits, then the compression ratio is calculated as the ratio of the compressed bit length to the uncompressed bit length:

$$\mu = \frac{C}{L \times \lg N}$$

The tabulation above displays this correct compression ratio as well as the observed decrease in the text (ASCII) file size.

It has been observed that for Turkish texts the alphabet for the Huffman coding includes *almost all* the possible syllables. No rule based on frequency values could be identified for exclusion from the alphabet. The mechanism that leads to inclusion/exclusion is heavily based on the way the dissolving affects the frequencies of the remaining alphabet members in favor of increasing/reducing the overall entropy. So, using a GA approach seems to be appropriate that suits the purpose of decision.

It would be interesting, as a future work, to consider the problem where the syllabification process is substituted by an iterative tri-gram process in which at each step tri-grams are attempted to be replaced by tokens according to a GA's decision. We believe that this will lead to a more generally applicable and more successful compression of any kind of data.

References

[1] L. D. Davis, *Handbook of Genetic Algorithms.* (Van Nonstrand Reinhold, 1991)

[2] D. E. Goldberg, *Genetic Algorithms* (Addison–Wesley Co., Reading, MA, 1989).

[3] D. E. Goldberg, Sizing Populations for Serial and Parallel Genetic Algorithms, *Proc. ICGA '89*, 70-79.

[4] R. W. Hamming, *Coding and Information Theory* (Prentice–Hall, Englewood Cliffs, NJ, 1986).

[5] T. Jones, *Evolutionary Algorithms, fitness Landscape and Search.* PhD thesis, (The University of New Mexico, New Mexico, 1995).

[6] K. A. De Jong and W. M.Spears, An Analysis of Multi-Point Crossover. *FGA*, 301-315, 1991.

[7] G. Lewis, *Turkish Grammar* (Oxford University Press, Oxford, 1991).

[8] S. Roman, *Coding and Information Theory* (Springer–Verlag, NY, 1992).

[9] R. Sedgewick, *Algorithms* (Addison–Wesley Co., Reading, MA, 1988).

[10] A. Wright, ed. *Foundations of Genetic Algorithms.* (Morgan-Kaufmann, 1991).