# **T-Code Compression for Arabic Computational Morphology**

Jim Yaghi and Mark R. Titchener

Department of Computer Science University of Auckland. jyag001@ec.auckland.ac.nz, mark@tcode.auckland.ac.nz

## Sane Yagi

Department of Linguistics University of Sharjah. saneyagi@yahoo.com

#### Abstract

It is impossible to perform root-based searching, concordancing, and grammar checking in Arabic without a method to match words with roots and vice versa. A comprehensive word list is essential for incremental searching, predictive SMS messaging, and spell checking, but due to the derivational and inflectional nature of Arabic, a comprehensive word list is taxing on storage space and access speed. This paper describes a method for compactly storing and efficiently accessing an extensive dictionary of Arabic words by their morphological properties and roots. Compression of the dictionary is based on T-Code encoding, which follows the Huffman encoding model. The special characteristics inherent in the recursive augmentation method with which codes are created allow compact storage on disk and in memory. They also facilitate the efficient use of bandwidth, for Arabic text transmission, over intranets and the Internet.

### 1 Introduction

### **1.1 Challenges of Arabic**

Arabic poses a formidable challenge for computational linguists due to its derivational nature. Word generation requires moulding three- and occasionally four-consonant roots into a range of morphosemantic template patterns, where the root radicals intersperse between a templates' letters to produce a new word with a new meaning that still shares the basic meaning of the root. Often these templates augment the root by lengthening its medial radical, inserting a long vowel between the radicals, and/or adding consonantal prefixes. The generated words are what is termed '*stems*' in the English language, but they are not actual words; they are mere semantic abstractions. To become actual words, the stems are moulded into morphosyntactic patterns that will indicate whether a word is a verb or noun, present or past, active or passive voice, etc.

The process of root extraction from actual words, on the other hand, is not a simple reversal of the process of word generation because the root radicals would have been disguised by the application of morphological patterns.

Although Arabic morphology is systematic, it has remained a challenge to produce, for example, useful spell checkers, grammar checkers, search engines, and indexers that are not based on exact matching. The missing ingredient at the base of this problem is an accurate root-based morphological analyser. Spell-checkers, normally, do not contain a large enough word list to accommodate the inflectional variation words undergo when affixed because these may run in millions. Few grammar checkers exist for Arabic, because it is difficult to parse a sentence if its words are not correctly interpreted by a morphological parser. Various types of Arabic search engines are significantly impaired because of the inability to find character-to-character correspondence between search terms and variant match items such as differing tense, voice, person, number, or gender.

### 1.2 T-Code Technique

Finite State Transducers (FSTs) have been established as a standard way to encode morphological alterations (Karttunen et al., 1997). However, FSTs are normally compiled from rules written in a special FST generator language. FST compilers like PC-KIMMO (Antworth, 1990) and 2lc (Karttunen, 1993; Karttunen and Beesley, 1992) use a specialised language to generate lexical transducers. On the other hand, our implementation uses the standard PERL regular expressions (Friedl, 2002) but in a specialised manner.

Beesley (2001) describes a system that generates FSTs using *2lc* for lexical transformations of Arabic words. When generating words, the system uses the compiled FSTs to achieve morphological and phonological letter alterations and then uses them in reverse to perform derivation. Our approach uses, like Beesley, the compiled FSTs for word generation, but it does not use it for root derivation.

Our approach produces a list of Arabic stems, inflected affixed forms along with their roots and complete morphological classifications; this list facilitates the direct regeneration of words. Our root derivation technique requires this extensive list or dictionary of stems to be stored in a searchfriendly manner.

The dictionary of stems would ordinarily occupy a large amount of disk storage space, but we propose here a technique that finds an acceptable balance between compression and lookup speed, T-Code (Titchener, 1984). T-Code compression is similar in style to Huffman encoding, as T-Codes are a subset of all possible Huffman code sets (Gunther, 1998). T-Code has the advantage of statistical synchronisation, or the ability to self-synchronise (Titchener, 1997), making it ideal for transmission over networks, especially where information loss is inevitable (eg., wireless networks).

We begin with basic roots and morphosemantic, morphosyntactic, and inflectional affixation rules to generate all possible stems. After some simple affix removal rules are applied, any valid lookup word should be found in the comprehensive list of stems. Internally, Arabic words are encoded with their roots and morphological classification so that the original word may be regenerated when needed.

In this paper, we discuss the system we have built for verb generation which can be used either in whole or in part for root-to-word and word-toroot lookups. We begin with a description of how the word list was generated, followed by a discussion of the dictionary format and how it was compressed, we then describe how the compressed dictionary was searched and decoded, and finally conclude with some suggestions for simple applications.

# 2 Word Generation

## 2.1 Word Data Creator Environment

The Word Data Creator Environment (WCE) was built to assist in creating and debugging the generation database. This software provides a graphical user interface facilitating data entry and experimentation.

WCE allowed us to edit the *MainDictionary Table*. For each entry, we were able to supply a root radical, a root classification identification number, and two numbers identifying the morphosemantic pattern and a morphosyntactic variant that a derived word would follow. Unlike traditional root-type classifications in Arabic morphology, our root classifications identify a root by the type and location of alterable letters it contains, for example, g(w), g(y), and  $\epsilon(')^1$ . Alterable letters are those that usually undergo rulebased transformation if followed or preceded by certain other letters.

In addition to entry editing, WCE allowed us to edit related template entries from the *Templates Table*. A *Templates Table* entry is indexed by a pattern and variant identifier and a tense and voice combination. Every entry specifies a general template string which, for the given voice and tense, causes derived words to have a certain meaning. Entries also identify a set of inflection and spelling transformation rules and an affix list number. Transformation rules are dependent

<sup>&</sup>lt;sup>1</sup>For readability, this paper uses Buckwalter's Arabic orthographical transliteration scheme (http://www.cis.upenn.edu/~cis639/arabic/info/translitchart.html).

on a combination of the template string letters and the root radicals. The template strings of each entry are, in fact, the combined result of a morphosemantic and a morphosyntactic pattern, transformed for the tense and voice of the entry. The possible tenses are past and present, the voices are passive and active, and the modes are indicative and imperative.

Affix lists, which were also editable from within WCE, contain patterns for generating 17 different morphosyntactic forms specifying combinations of gender, number, and person for each voice and tense. Both affixation and transformation rules are specified using the language of PERL regular expressions.

#### 2.2 Word Generation Engine

Within WCE is an implementation of the Word Generation Engine (WGE), which allowed us to debug our classifications and transformation rules, and to ensure the correct spelling of generated words. While making modifications to root radicals, word classifications, template strings, and transformation and affixation rules, we were able to preview the result of any of the 17 word types on the main screen for the selected *MainDictionary Table* entry.

The three components, *Stem Transformer*, *Affixer*, and *Slotter*, activated in sequence, make up WGE. *Stem Transformer* applies the appropriate transformation rules to the stem template, *Affixer* adds an affix to the transformed template, and finally *Slotter* applies the transformed radicals to the affixed template to produce the final affixed word.

WGE begins with a stem ID from the *MainDictionary Table* as input. The root and entry associated with the stem ID are used to identify the radicals of the root, the stem template string, the set of transformation rules that apply, and an affix list.

*Stem Transformer* is applied incrementally using radicals, a template string, and one transformation rule per pass, as in Figure 1. The output of each pass is fed back into *Stem Transformer* as a modified template string and modified radicals, along with the next transformation rule. When all rules associated with the template are exhausted, the resultant template string and radicals are output to the next phase.

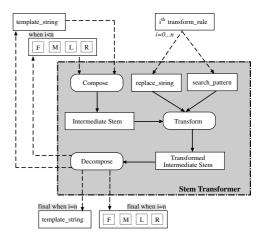


Figure 1: Stem Transformation Phase

A template string marks the positions at which radicals belong in the template by using the Roman letters F, M, L, and R. These may be viewed as the variables in the template; all other characters are Arabic constants. Stem Transformer begins by inserting the root radicals directly after their position markers. For example, a template, ا ت  $M L^{(iFotaMaLa)^2}$ , with a radical set رَل كَ M تَ ذْ ال ال ({\*,k,r}), becomes ({ ذ، ك، ر} (<iF\*otaMkaLra). The result is then decomposed back into the template form, and the root radicals are updated if altered. For the same example, the stem template is transformed by an ordered sequence of rules  $\{1,12\}$ . The text of rule 1 (ت)(\* [ ْ ْ ْ ْ ]\*)(ت is: F\$1\$2\$1. The first part specifies the match pattern and the second specifies the replace string. Rule 1 removes the infix letter ت (t) and replaces it with a copy of the first radical which should directly follow the radical's diacritic. The result is the string, ا کَ Míd L (<i**F**\*o\*a**M**kaLra). رُط کَ ا

Stem Transformer concludes by decomposing the updated template into template text and radicals. The altered template and radical set are then passed back into Stem Transformer, where another rule from the rule sequence may be applied. For the example above, the decomposed template becomes  $\downarrow F^{\circ} \stackrel{\circ}{\searrow} M^{\circ} L^{\circ}$ (<iFo\*aMaLa), while the root radical set remains unchanged. During this second pass, Stem Transformer uses the altered template and radical set

 $<sup>^{2}</sup>$ The letters F, M, L, and R in bold are radical position markers, not transliterations.

as input together with rule 12, whose text is, ([FMLR]?)([^ ' ' ]\*)([ ' ' ]\*)([FMLR]?) (\2) \$1\$2'. Ŕule 12 is a gemination rule, and uses a backreference in the search pattern, in order to match any repeated letter. With its replace string, the second of the duplicate letters is replaced by the gemination diacritic, / '/ (~). The decomposed result is the template, |F'ML', and the untransformed radical set,  $\{j, L, k, r\}$ ), which can produce the word  $|\vec{L}| < |\vec{L}| < |\vec{L}|$ .

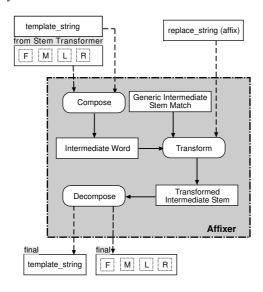


Figure 2: The Affixer Phase

In brief, the final output of Stem Transformer is a root-transformed template and a templatetransformed radical set. These outputs are used as input to the affixation phase which succeeds stem transformation. Affixer, which is applied iteratively on the result of Stem Transformer, outputs 17 different affixed morphosyntactic forms for every stem. Affixer is run with different replace strings specific to the type of affix being produced. It modifies copies of the transformed stem from the previous phase, as in Figure 2. For example,  $\downarrow F M L$  is passed to Affixer, with radical set,  $\{ :, :, :\}$  ({\*,k,r}), and the past active feminine singular affix replace string, \$1\$6M\$7´L\$11`ت. Figure 3 shows the generic transformed-template match string and indicates the back-reference groupings, which are used in the replace string for the affix. The result of applying the affix transformation above is the affixed template string, إ F M L ت (<iF~aMaLato).

F	M	
2 (([^FMLR]*) F ((´´, `)*) ((´, *))	6 (([^FMLR]*) M ((´´´`)*) ((´)`)	
1	5	
L	R	
10 (([^FMLR]*) L ((´´``)*) ((´`*`))	14 (([^FMLR]*) R ((´´´`)*) ((´´')*)	

Figure 3: The generic transformed-template match string

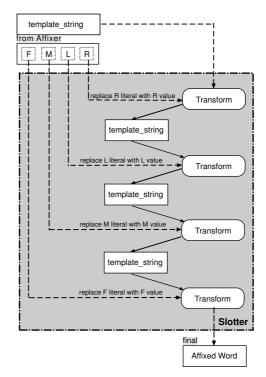


Figure 4: The Slotter Phase

In *Slotter*, the last stage of word generation, transformed radicals replace the Roman position markers in the transformed template, to produce the final form of the word. For the example above, the result is  $(<i^*-akarato)$  which is the past active feminine singular form of the word.

### 3 T-Code Encoding

Using a format that allows searching the database, we output an alphabetically sorted list of each of the 2 million words that WGE generated. Since diacritics are optional in written Arabic, we wanted to facilitate the matching process by having the possibility of ignoring diacritics or only matching those diacritics that a search item specifies. In order to achieve this, we indexed our list for lookup using bare words, words without diacritics. For each entry, we included the root, template, and affix type identifiers as numbers. This gave the capability of generating the actual word after lookup in order to pinpoint an exact diacritic match if necessary.

Indexing the complete word list from WGE and storing it in a disk-based B-Tree data structure yields files larger than 100 MB<sup>3</sup> Since our dictionary only represents the verbs of Arabic, adding the nouns would at least double its size; therefore, it would be advantageous to keep the dictionary's disk size minimal.

T-Code encoding, like Huffman encoding, is a variable length coding scheme. The basis of T-Code text compression is that shorter codes are assigned to frequently repeated items. Since uncompressed text is normally represented by fixed length codes in software, T-Code is capable of achieving a large compression factor for text because it has low entropy. For the word database produced by WGE, a great amount of redundancy exists since the 2 million words are based on only 5,500 verb roots. T-Code has the advantage of self-synchronisation; that is, a series of bits from a code will only be recognised as being members of the T-Code set if they constitute a valid code word. If a series of bits does not belong to the T-Code set, it will not be valid until all the bits of the code arrive. This is useful because no additional code length information needs to be stored in the data.

The T-Codes used to encode the database are obtained by first calculating a target distribution of code bit-lengths, then creating an adjusted T-Code distribution based on the target, and finally assigning the shortest codes to the most frequent data items.

#### 3.1 Calculating a Target Distribution

A target distribution for the dictionary database was calculated using the frequencies of its unique items. The equation below was used to calculate the code's target bit-length  $\ell$  for each data item *i* 

in the database using the item's average frequency  $\bar{f}_i$ .

$$\ell_i = -\left\lceil \log_2 \bar{f}_i \right\rceil, i = 0...n$$

We grouped the frequencies of unique root, template, and affix type identifier numbers for each word entry. Additionally, a slightly different frequency count for the letters of the lookup words was performed in order to take into account their compressed form. Special attention was given to the compression of the low entropy lookup words whose efficient access is essential.

Original		Letters Counted	
Entry	Transliteration	Entry	Transliteration
اب	Ab	اب	Ab
ابا	AbA	۱	A
ابابا	AbAbAبا		bA
اباتت	AbAtt	تت	tt
اباتمتم	AbAtmtm	متم	mtm
اباتنتن	AbAtntn	نتن	ntn
اباث	AbAv	ث	v
اباثن	AbAvn	ن	n
اباج	AbAj	ع	j
اباجا	AbAjA	l	A
اباجن	AbAjn	ن	n
باجوا	bAjwA	باجوا	bAjwA

Table 1: Eliminating redundancy by not counting repeated letters.

The bare words forming the lookup entries have a one-to-many relationship with actual words. That is, many different generated words with diacritics may become the same lookup entry when diacritics are removed. Therefore, if it is possible to distinguish between one bare word and the next, repetition of lookup entries is unnecessary. A bit-skip field is used in the encoded database to mark the end of an entry; details of this and the encoded database format are discussed in Section 3.2. During this phase, we were only concerned with the frequency of the letters of the lookup items in the final database, so unique entries had their letters counted only once.

Another source of redundancy in lookup items appears in their alphabetically sorted form. Often, an entry shares initial letters with following

<sup>&</sup>lt;sup>3</sup>The file size being so large is explainable by the fact that Unicode UTF-16 uses 16-bits per Arabic character (Consortium, 2003), which causes output to be twice as large as it may have been for Roman characters.

entries. While the dictionary format handles this, calculation of a target distribution only counts letters not sequentially shared between consecutive entries, as may be seen in Table 1.

Code	Code Target Modified		T-Code	
Length	Frequency	Target	Distribution	
5	9	5	5	
6	5	9	9	
7	-	-	0	
8	2	2	2	
9	-	-	0	
10	4	4	4	
11	3	3	3	
12	9	9	9	
13	10	10	10	
14	2	2	2	
15	-	-	0	
16	16378	3836	3836	
17	2750	7232	7233	
18	-	8059	14159	
19	-	-	27308	
20	3	3	52009	

Table 2: A T-Code distribution from the target distribution for the dictionary.

#### **3.2** Encoding the Dictionary

A T-Code distribution was calculated based on the target distribution, as in Table 2. Its codes were created and sorted from shortest to longest then assigned to the unique data items of the database in order of most frequent to least frequent. The uncompressed dictionary's data items were then T-Code encoded.

Figure 5 depicts the encoded dictionary structure. A header is used to identify the positions of the start and end of the encoded data. The T-Code encoded data is represented as a continuous bit stream written in byte-sized units.

#### 3.2.1 Indexing and Accessing the Dictionary

Access to the dictionary is required to be sequential. Without a proper indexing system lookups would be inefficient, having potential complexity of order O(N). To facilitate efficient lookups, a simple first letter lookup was used to give direct access to the byte position of the first entry using the first letter of the lookup word.

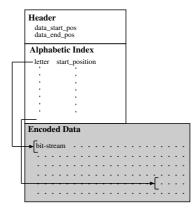


Figure 5: The encoded dictionary structure.

While the first-letter-lookup gives a reasonable efficiency advantage, the rest of the lookup process is required to sequentially read the entries starting with the first letter. In order to address this, we added a two-byte fixed width field at the start of every entry, and distributed their bits as in Figure 6. An example in Table 3 illustrates how the fixed width fields are used.

Pos.	Entry	Transliteration	Next Pos.	Shared
0	اب	Ab	11	0
1	ابا	AbA	11	2
2	ابابا	AbAbA	3	3
3	اباتت	AbAtt	4	3
4	اباتمتم	AbAtmtm	5	4
5	اباتنتن	AbAtntn	6	4
6	اباث	AbAv	8	3
7	اباثن	AbAvn	8	4
8	اباج	AbAj	11	3
9	اباجا	AbAjA	10	4
10	اباجن	AbAjn	11	4
11	باجوا	bAjwA	12	4

Table 3: An example illustrating how the next entry bit-skip and shared letter fields are used.

The first 12 bits store the distance in bits to the next test entry. If the word being searched for in the dictionary does not have a partial match with the test word at the current entry, the bit-skip field points to the next entry that does not begin with all the same letters. If a partial match is found, then only words between the current position and the bit-skip position may match the lookup word.

The remaining 4 bits store information on the

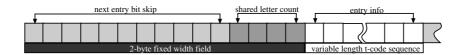


Figure 6: An entry using 12 bits for number of bits to skip to next entry and 4 bits for the number of shared letters.

number of letters shared between the current word and the next word. This allows the decoder to compare only the codes of the letters that have not been tested earlier, reducing the number of comparisons needed to make a match.

#### 3.2.2 Results

Using T-Codes and the indexing system described in this section, the dictionary disk-size was reduced to a mere 8 megabytes. The current dictionary size includes search and lookup information, which is over 90% smaller than the uncompressed B-Tree version with a comprable lookup speed.

Two devices may use a copy of the dictionary in order to communicate using T-Code transmission. A device may encode and transmit every Arabic word in a message into three codes containing the root, template, and affix identifiers for the word. The bandwidth used to transmit an Arabic word becomes a fraction of the equivalent T-Code encoded word.

بَكتُبونَ For example, consider a word such as (yaktubwna), which consists of the root, template, and affix identifier set {12884,460,30}. The T-Code lengths will depend only on the statistical frequency of each of these identifiers for all the words in an Arabic corpus so as to provide maximum efficiency; in this case the word may be represented as {0010101, 001001, 10100} and transmitted as 18 bits. Compare this size with the same word transmitted in Unicode. This 9 letter word would normally require 2,592 bits to be transmitted in raw Unicode(16-bit per character x 9 characters). If, instead, the raw identifier set was transmitted, it would require 48 bits (16-bit per integer x 3 integers), which is still significantly higher than the T-Code encoded form.

# 4 A Simple Application: A Root Extractor and Word Parser

To demonstrate the efficiency of the dictionary, we created a PERL based implementation of the decoder, and wrote a web CGI that derives and parses Arabic words. This particular implementation, although very simple, also functions as an accurate root extractor.

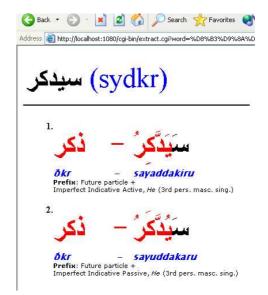


Figure 7: Example output from the word-parser Web CGI using the T-Code encoded dictionary of Arabic words.

A UTF-8 Unicode-encoded HTML webpage accepts Arabic words in a simple form. The CGI is invoked with the input stripped of diacritics. Next, the CGI removes combinations of conjunction, prefix, and suffix letters that it finds in a pre-supplied list of affixes and it begins with the longest to the shortest sequences. The original word and each of its stripped forms are T-Code encoded and pushed into a queue. Entry codes that match any of the items in the queue are retrieved with their identifier lists from the dictionary and decoded. Identifiers are used in order to generate the words with diacritics that the entry identifies. Also, the identifier information is used to morphologically classify the entered word and the affixes that are used with it. The various possible morphological parsings are then output to HTML, as in Figure 7.

## 5 Further Work

We have described a system that uses T-Code to compress and access a comprehensive list of Arabic verbs by their morphological properties. Word generation here is restricted to verbs, but further research must extend the coverage to verbs and rootless words such as particles and loan words.

Once data has been obtained for word generation of nouns, the implementations of many of the applications discussed in the introduction become feasible. For example, a spell checker can be instructed to recognise conjunction, prefix, and suffix letter combinations, as described in Section 4. Since these letters do not cause alteration to adjacent letters, they may be removed and the remaining stem looked up in the dictionary. If a match is not found, a spelling error may be reported. Suggested spellings may come from the word-generation and transformation rules of the closest matching word or words. The closest match, like in English spell-checkers, would be the words that have reasonable character-correspondence.

Using the root-extraction algorithm in Section 4, root-based searching becomes possible. Both the search term and search text will undergo root extraction before a match is found.

Incremental searches such as that used in predictive text messaging only need to have a list of the conjunctions and affixes added to the dictionary list. The implementation can then allow combinations of conjunctions and affixes to attach to dictionary entries. Since the dictionary list now includes all forms affixed, transformed, and disguised, valid Arabic words will always find a match in the dictionary.

In the near future, we hope to increase the lookup and decoding speed by creating a T-Code Finite State Automaton (FSA) for the dictionary as described in (Nithyaganesh, 1998), which will be able to read an entire byte or two and output several code words. Currently, the decoding process tests if a code belongs to the T-Code set; if it does not match, another bit is added to the T-Code before it is tested once more. This continues until the code matches a code from the valid T-Code set. With a T-Code FSA, a significant improvement in the decoding speed will be witnessed, since bytes are looked up rather than bits.

### References

- Evan L Antworth. 1990. PC-KIMMO: a two-level processor for morphological analysis. *Occasional Publications in Academic Computing*, 16.
- Kenneth R Beesley. 2001. Finite-state morphological analysis and generation of arabic at xerox research: Status and plans in 2001. In *ARABIC Language Processing: Status and Prospects*. Arabic NLP Workshop at ACL/EACL 2001, July.
- The Unicode Consortium, 2003. *The Unicode Standard, Version 4.0*, chapter 2, page 29. Addison-Wesley, Reading, MA. ISBN 0-321-18578-1.
- Jeffery E. F. Friedl. 2002. *Mastering Regular Expressions*. O'Reilly, 2nd edition, July.
- Ulrich Gunther. 1998. *Robust Source Coding With Generalised T-Codes*. Ph.D. thesis, University of Auckland.
- Lauri Karttunen and Kenneth R Beesley. 1992. Twolevel rule compiler. Technical Report ISTL-92-2, Xerox, Xerox Palo Alto Research Center, Palo Alto, California.
- L. Karttunen, J-P. Chanod, G. Grefenstette, and A. Schiller. 1997. Regular expressions for language engineering. In *Natural Language Engineering*, pages 238–305. February 5.
- Lauri Karttunen. 1993. Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox, Xerox Palo Alto Research Center, Palo Alto, California, April.
- Kirubalaratnam Nithyaganesh. 1998. *The Talk-Net Project Real-Time Speech Communication Using T-Codes*. MSc thesis, University of Auckland.
- Mark R Titchener. 1984. Digital encoding by means of new t-codes to provide improved data synchronization and message integrity. In *Technical Note, IEE Proceedings*, volume 131 of 4, pages 51–53, July.
- Mark R Titchener. 1997. The synchronization of variable-length codes. *IEEE Transactions on Information Theory*, 43:683–691, March.