ACB Compression Method and Query Preprocessing in Text Retrieval Systems

Tomáš Skopal

VŠB-Technical University of Ostrava, Computer Science Dept.

e-mail: Tomas.Skopal@vsb.cz

Abstract. This article presents ACB – high efficient text compression method, its word-based modification and finally discuss relating benefits by creating auxiliary query subsystem for usage in text retrieval systems.

Key words: ACB, compression, context item, index, query preprocessing

1 Compression and Indexing in TRS

Data compression is widely used in text (or full-text) databases to save storage space and network bandwidth. In typical full-text database, various auxiliary structures are provided in addition to the main compressed text. They include at least a text index, a lexicon, and disk mappings [BELL93].

Many compression methods can be used to reduce size of all these data. Relatively more advanced methods are based on coding words as basic unit. We could cite word-based adaptive Huffman coding and HuffWord [HC92, WIT94], another experiments have been done with word-based LZW by Horspool and Cormack [HC92, SAL98].

However, word-based compression methods may serve moreover for other purposes, in addition to data compression itself. One of these purposes have been presented by Dvorský, Pokorný and Snášel in [ADBIS99]. Their modification of LZW (WLZW) simultaneously provides an indexing of source text (documents). In addition to the compressed data stream, a token index file is produced. This file maintains for every token an inverted entry, which contains a list of all documents where the token occurs.

Thus, index file is particular example of interconnection between data compression and TRS. Another approach of such interconnection is introduced in the following section.

2 Query preprocessing

In TR environment we often don't know how to query for intended data. User knows only few (and/or very general) terms, which can be found in very many documents.

On the other side, user may express the conceptual content of the required information with query terms which do not match the terms appearing in the relevant documents.

This vocabulary problem, discussed, for example, by Furnas et al. [FUR87], is more severe when the user queries are short or when the database to be searched is large, as in the case of Web-based retrieval.

A survey of this topic is thoroughly presented in [ACM01] where several query-advancing techniques are discussed, leading to relevant documents response.

Back to the data compression. The word-based modification of ACB compression method is suitable for production (besides compressed stream and text index) of structure (i.e. context index) which can be used as a standalone TR tool for query advancing. This structure allows us to query upon data we don't exactly know, using a simple term-matching query. As a response of that query, we will obtain a set of word-based substrings contained within indexed/compressed documents, in which the original query term appears. These substrings we call **data contexts**. Furthermore, in the data context we can browse for any other terms lexicographically/semantically connected with the original term. Such sequences of iterative querying-obtaining will produce more terms, even term strings, which can be issued to the standard full-text searching systems (through vector/boolean queries).

Thus, idea of data contexts can be utilized for advanced query construction. Following section will introduce the fundamental structures of ACB method, respectively structures for building context-based TR subsystem. Detailed description of plain ACB method can be found in [SAL98].

3 Basic structures

In following definitions we will use general strings – sequences of terms (characters or tokens) – and the common string operations. Representation of terms can vary from bits through ASCII characters to general tokens.

3.1 Context, content, context item

Every term (character/token) t_i in the given text can be represented as a position in the text. Let any substring α in the left vicinity from this position is called **context** and any substring β in the right vicinity (including t_i) $\beta = t_i.\beta_{ext}$ is called **content** of term t_i. Let the pair $I_i = (\alpha, \beta)$ be **context item** of term t_i. Let γ_j be the substring (term string) of the text, $\beta = \gamma_j.\beta_{ext}$, then pair $J_j = (\alpha, \beta)$ is context item of string γ_j. If context or content of context item is limited by a number of terms, then the context item is bound.

Example 1.

a) Terms are ASCII characters
text: `swiss␣miss␣is␣missing`
4-bound context item for the 4th 's' is '`s␣mi`$\underline{\texttt{s}}$`s␣i`'

b) Terms are bit characters
text: `10011101011`
unbound context item for the 3^{rd} 0 is '`10011`**`0`**`1011`'

c) Terms are words (identifiers)
text: `'swiss''miss''is''missing'`
2-bound context item for the 'is' is `''swiss''miss'`'is'`'missing''`

Any subset of all possible k-bound context items of the text form k-bound **context item collection**.

3.2 Context dictionary

Structure of **context dictionary** adds a complete order to the context item collection. The order can be described by following:

1. Context item I is smaller then context item J if α_I is smaller using **reverse comparison** then α_J. Reverse comparison means right-to-left. Comparison of individual terms is interpretation-dependent.
   Formally holds: $\alpha_I < \alpha_J \rightarrow I < J$

2. If both contexts are equal, contents are examined using **regular comparison** (left-to-right).
   Formally holds: $(\alpha_I = \alpha_J)\&(\beta_I < \beta_J) \rightarrow I < J$

3. If both contexts and both contents are equal then context items are equal.

Duplicate items are not allowed.

```
1 ...swiss␣|m.....
2 ......swi|ss␣m..
3 ........s|wiss␣m
4 .....swis|s␣m...
5 ....swiss|␣m....
6 .......sw|iss␣m.
```

Fig. 1: Context dictionary, terms are ASCII chars (vertical separates context and content)

4 Compression method ACB

This compression method[1] uses idem structures for highly efficient text compression. Compression algorithm passes the text term-by-term obtaining context items by usual fashion. At the beginning of encoding/decoding, the runtime[2] context dictionary is empty.

4.1 Encoding algorithm

1. Current context item C (made up from text already encoded (context) and the rest of the text (content)) is examined against the actual state of context dictionary. Position i of the nearest context item N (most similar to C) in the dictionary is found using the defined order, but comparing only the contexts. Formally holds: $(\alpha_C \leq \alpha_N) \& \forall K(\alpha_K < \alpha_N \rightarrow \alpha_K < \alpha_C)$

2. From this position in dictionary we incrementally search (in both directions – the search radius is naturally limited) a context item M at a position k, where β_M best match β_C. Best match is determined by n – maximal count of equal terms.

3. To the output goes a triplet $(k-i,n,t)$ – where t is the first non-matching term in β_C (from step 2)

4. The context dictionary is updated with $n+1$ new context items. Each new added context item is the previous one incremented by 1 (i.e. the context is longer by 1 and the content is shorter by 1).

Example 2. (encoding step) :

Text being encoded is string from example 1a. Dictionary's actual state is shown at the figure 1. Current context item is `'swiss␣m|iss␣is␣missing'`.

1. Appropriate item in context dictionary is found. Item at position 2.

2. Then search in the dictionary from this position for item with best content match. Item at position 6. 4 chars match (`'iss␣'`), first non-matching char in the current item content is `'i'`.

3. To the output goes triplet (4,4,`'i'`).

4. Context dictionary is updated with 5 new items (see figure 2).

[1]Associative Coder of Buyanovsky by George Buyanovsky

[2]means in-memory; items are positions of terms and are unbound

```
1  ...swiss␣miss␣|i..........
2  ........swiss␣|miss␣i.....
3  ......swiss␣mi|ss␣i.......
4  ...........swi|ss␣miss␣i..
5  .......swiss␣m|iss␣i......
6  .............s|wiss␣miss␣i
7  .....swiss␣mis|s␣i........
8  ..........swis|s␣miss␣i...
9  ....swiss␣miss|␣i.........
10.........swiss|␣miss␣i....
11............sw|iss␣miss␣i.
```

Fig. 2: Context dictionary after update (items in frame are the new ones)

4.2 Decoding algorithm

The decoding is exactly inverse. Runtime context dictionary grows the same way as by encoding. Text is reconstructed applying triplets on the actual state of dictionary.

Example 3. (decoding step) :

Initial current item and state of dictionary are the same as in previous example. Input triplet (4,4,`'i'`) is processed by following:

1. Find in dictionary the appropriate item for current item. Same way as by encoding. Item 2 was found.

2. Take the first member of triplet (here 4) and add it to the found position. Get item on position 6. Read from the items content first n chars specified by the second triplet member. You get `'iss␣'`.

3. To the output goes `'iss␣'` + the third member `'i'` $\rightarrow$ `'iss␣i'`

4. Dictionary is updated with 5 new items (see figure 2)

4.3 Benefits of ACB brought into TRS

- ACB uses for each file (data stream) new dictionary, which serves as a compression tool. This data-locality allows represent context by numbers (positions to source text).

- For purposes of TRS we can persist produced dictionary as a valuable source of semantic relations. For a growing collection of documents there could evolve big dictionary over time. In this phase we should speak rather about **context index** than about persistent dictionary.

- ACB algorithm itself can provide (as a side effect) heuristic techniques to improve context index creation and analysis.

5 Building context-based TR subsystem

General context index conception must now become more clear. Because of context index is persistent and standalone structure, context items cannot be represented as some positions in source text. This feature considerably differs from runtime context dictionary which can be unbound. Context index items are represented as term identifier strings. Identifiers and terms are associated in text index which serves as a lexicon as well.

Thus, context index must be bound and length of context items significantly determines storage costs. Fortunately, lengths up to 10 terms seem to be sufficient.

5.1 Simple query

Example 4. (one item in 4-bound context index) :

```
              :
              :
steam engines | work reliably
              :
              :
```

When querying on 'engines', string "'steam' 'engines' 'work' 'reliably'" will return as a response. This answer offers possibility that the word 'engines' relates *somehow* to the other words. The user have obtained more information and is able to formulate better query.

5.2 Context item reduction

It is obvious that even 10-bound context index might produce big storage overhead. Linguistically, only few words are possible to be semantically interconnected with words in distance greater than 3. However, this is a question for additional testing in real TR environment.

5.3 Alphabet of terms

Next goal in the process of context index reduction is the alphabet reduction. This can be achieved by following restrictions:

- Word and non-word consideration as presented in [ADBIS99]. Words are semantic terms (words) and non-words are blank terms (spaces, separators, tabs, etc). Words and non-words strictly alternate (word is immediately followed by non-word).

Context index contains only words (i.e. their identifiers). This feature considerably improves response relevancy.

- Reduction achieved by *lemmatizers*[3]. Lemma is the lexical root of its various derivates (for example 'go' is lemma for 'goes','went','going',etc). Instead of that derivates, the lemma will be used in context index. However, this loss of information may worsen the response relevancy.

- Ignoring stop-words consideration. Stop-word is a word with minimal semantic meaning. For example 'the', 'a', 'it', 'that'.

Reduction of alphabet causes reduction of context index as well. There will be high probability (even higher with low-bound items), that required item addition will produce duplicate items in the index – that is not allowed – thus the item will be ignored.

6 Future work

- In the future we are going to examine **transitive queries**, i.e. complex queries combining particular context-content match.

  Example 5.

  ```
  steam engine | is an ancestor of jet propulsion
                                   jet propulsion | uses liquid fuel
  ```

  Request on 'steam engine' might produce 'jet propulsion' or even 'liquid fuel'.

- Nowadays, all the presented stuff is under software development and extensive testing.

Rereferences

[ADBIS99] J. Dvorský, J. Pokorný, V. Snášel: *Word-based Compression Methods and Indexing for Text Retrieval Systems*, Proc. ADBIS'99, Springer Verlag, 1999, pp. 75-84

[ACM01] C. Carpineto, R. de Mori, G. Romano, B. Bigi: *An Information-Theoretic Approach to Automatic Query Expansion*, ACM Transactions and Information Systems, Vol. 19, No. 1, January 2001

[FUR87] G.W. Furnas, T.K. Landauer, L.M. Gomez, S.T. Dumais: *The vocabulary problem in human-system communication.* Commun. ACM 30, 11 (Nov.) 1987, 964971.

[3]stemming techniques

[BELL93] T.C. Bell et al: *Data Compression in Full-Text Retrieval Systems*, Journal of the American Society for Information Science. 44(9), 1993, pp.508-531

[HC92] R.N. Horspool, G.V. Cormack: *Construction Word-based Text Compression Algorithms*, Proc. 2nd IEEE Data Compression Conference, Snowbird, 1992

[SAL98] D. Salomon: *Data Compression*, Springer Verlag, 1998

[WIT94] I.H. Witten, A. Moffat, T.C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images.*, Van Nostrand Reinhold, 1994