The Treegram Index—An Efficient Technique for Retrieval in Linguistic Treebanks

Hans Argenton and **Anke Feldhaus**

Infineon Technologies, DAT CIF, Postbox 801709, D-81617 München
hans.argenton@infineon.com
University of Tübingen, SfS, Kleine Wilhelmstr.113, D-72074 Tübingen
feldhaus@sfs.nphil.uni-tuebingen.de

Multiway trees (MT, henceforth) are a common and well-understood data structure for describing hierarchical linguistic information. With the availability of large treebanks, retrieval techniques for highly structured data now become essential. In this contribution, we investigate the efficient retrieval of MT structures at the cost of a complex index—the *Treegram Index*.

We illustrate our approach with the VENONA retrieval system, which handles the BH^t (Biblia Hebraica transcripta) treebank comprising 508,650 phrase structure trees with maximum degree eight and maximum height 17, containing altogether 3.3 million Old-Hebrew words.

1 Multiway-tree retrieval based on treegrams

The base entities of the tree-retrieval problem for positional MTs are (labeled) rooted MTs where children are distinguished by their position.

Let s and t be two MTs; t *contains* s (written as $s \preceq t$) if there exists an injective embedding such that (1) nodes are mapped to nodes with identical labels and (2) a root of a child with position i is mapped to a root of a child with the same position.

Retrieval problem: Let DB be a set of labeled positional MTs and let q be a query tree having the same label alphabet. The problem is to find efficiently all trees $t \in$ DB that contain q.

To cope with this tree-retrieval problem, we generalize the well-known n-gram indexing technique for text databases: In place of substrings with fixed length, we use subtrees with fixed maximal height—*treegrams*.

Let TG(t,h) denote the set of all treegrams of height h contained in the MT t, and let $\mathrm{T}(\mathrm{DB}, g)$ denote the set of all database trees that contain the treegram g. Assume that g has the height h and that $\mathrm{T}(\mathrm{DB}, g)$ can be efficiently computed using the index relation $I^h_{DB} := \{(g,t) | t \in \mathrm{DB} \wedge g \in \mathrm{TG}(t,h)\}$, which lists for each treegram g of height h every database tree that contains g. We compute the desired result set $R = \{t \in \mathrm{DB} | q \preceq t\}$ for a given query tree q such that q's height is greater than or equal h as follows:

Retrieval method:

(1) Compute the set $\mathrm{TG}(q,h)$: All treegrams of height h contained in the query.

(2) Compute the *candidate set* of q $\mathrm{Cand}_h(q) := \bigcap_{g \in \mathrm{TG}(q,h)} \mathrm{T}(\mathrm{DB}, g)$. The set of all database trees that contain every query treegram.

(3) Compute the *result set* $R = \{t \in \mathrm{Cand}_h(q) | q \preceq t\}$.

The costly operation in this approach is the last containment test $q \preceq t$. The building of index I^h_{DB} is justified if in general the

number of candidates will be much smaller than the number of trees in DB.

2 Efficient query evaluation

The treegram-index retrieval method given above encounters the following interesting problems:

(A) A single treegram may be very complex because of its unlimited degree and label strings; this leads to costly look-up operations.

(B) There are many treegrams rooting at a given node in a database tree: To accomodate queries with subtree variables, the index has to contain all matching treegrams for that subtree.

(C) It is quite expensive to intersect the tree sets $\mathrm{T}(\mathrm{DB}, g)$ for all treegrams g contained in the query q.

VENONA addresses these problems by the following approach:

Problem A: *Processing of a single treegram:* (1) Node labels hash to an integer of a few bytes: We do not consider labels structured; to model the structure of word forms, feature terms should be used[1]. (2) VENONA deals only with treegrams of a maximal degree d; if a tree is of greater degree, it will be transformed automatically to a d-ary tree.[2] (3) For describing a single treegram g, VENONA takes each of g's hashed labels and combines it with the position of its corresponding node in a complete d-ary tree; an integer encoding g's structure completes this representation: Structure is at least as essential for tree retrieval as label information.

[1] Due to lack of space, we cannot present our extension of treegram indexing to feature terms in this abstract.

[2] The employed algorithm is a generalization of the well-known transformation of trees to binary trees. d's value is a configurable parameter of the index-generation.

Problem B VENONA uses only one treegram per node v: the treegram including *every* node found on the first h levels of the subtree rooted in v. This approach keeps the index small but introduces another problem: A query treegram may not appear in the treegram index as it is. Therefore, VENONA expands all query treegram *structures* at runtime; for a given query treegram g, this expansion yields all database treegrams with a structure compatible to g. That approach keeps the treegram index small and preserves efficiency.

Problem C The evaluation of a given query q is processed along the following steps: (1) According to q's degree and height, VENONA chooses a treegram index among those available for the tree database. (2) VENONA collects q's treegrams and represents them by sets of treegram parts. For a given query treegram, VENONA expands the structure number to a set of index treegram structures and removes those labels that consist of a variable: Variables and the constraints that they impose belong to the matching phase. (3) VENONA sorts q's treegrams according to their selectivity by estimating a treegram's selectivity based on the selectivity of its treegram parts. (4) VENONA estimates how many query treegrams it has to evaluate to yield a candidate set small enough for the tree matcher; only for those it determines the corresponding index treegrams. (5) VENONA processes these selected treegrams until the candidate set has the desired size—if necessary, falling back on some of the treegrams put aside. (6) Finally, the tree matcher selects the answer trees from these candidates.