

- Pattern Matching (continued)
  1. Ukkonen's Algorithm for more efficient dynamic programming
- Introduction to Indexing and Searching
- Inverted file indexing
  1. Compressing the posting lists
  2. Representing and accessing the lexicon

1 Pattern Matching (continued)

So far we have discussed algorithms that take at least $O(nm)$ time for two sequences of length n and m. In an application such as `diff` this can be prohibitively expensive. If we had two files of $100,000$ lines each, the algorithm would require 10^{10} string compares, which would take hours. However, `diff` can actually run much faster than this when the files to be compared are similar (*i.e.*, the typical situation in which they both come from the same source and just involve minor changes).

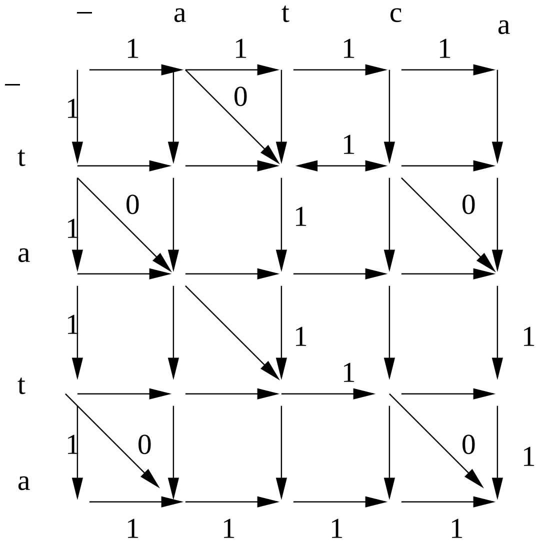
We will now consider an algorithm for edit-distance which takes $O(\min(n,m)D)$ time, where D is the edit distance between the two strings. The runtime is therefore output dependent. This algorithm can also be used for the longest common subsequence and a variant is used in the `diff` program.

1.1 Ukkonen's Algorithm

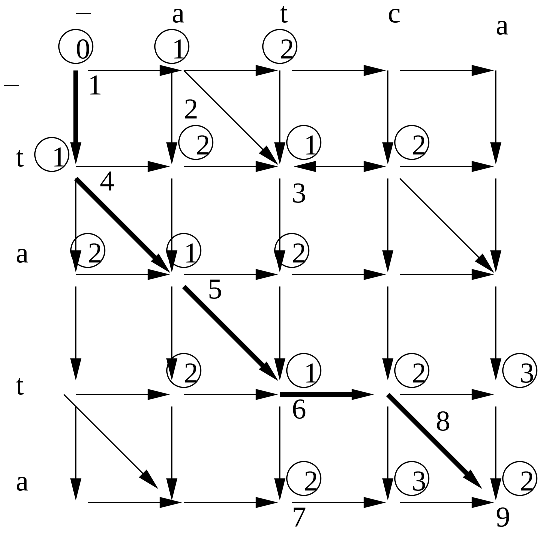
Consider the minimum edit distance problem:

$$\begin{aligned} D_{i0} &= i \\ D_{0j} &= j \\ D_{ij} &= \begin{cases} D_{i-1,j-1} & a_i = b_i \\ 1 + min(D_{i,j-1}, D_{i-1,j}) & \text{otherwise} \end{cases} \end{aligned} \tag{8}$$

This problem can be viewed as a weighted directed graph laid over the $n \times m$ matrix of $D_{i,j}$. It can then be solved by finding a shortest path in the graph from $D_{0,0}$ to $D_{n,m}$. This graph has a unit weight edge between neighbors in the same row or column. These edges represent an insertion or deletion—the $1+min(D_{i,j-1}, D_{i-1,j})$ case in Equation 8. The graph also has zero weight diagonal edges from $D_{i-1,j-1}$ to $D_{i,j}$ whenever $a_i = b_i$—representing the $a_i = b_i$ case in Equation 8. Figure 134(a) shows an example of such a graph. In this graph the length of a path represents the cost of those set of modifications. Therefore the shortest path from $D_{0,0}$ to $D_{n,m}$ gives the set of changes for the minimum edit distance. Note that we don't actually have to construct the graph since we can determine the edges from a location i, j simply by looking at a_i, a_{i+1}, b_j and b_{j+1}.



(a) Graph



(b) Shortest Path

Figure 134: Graph representation of the minimum edit distance problem for the strings `atca` and `tata`. In the right figure the circled values specify the distances and the uncircled values specify the visiting order for Dijkstra's algorithm. The shaded path represents the shortest path.

To find the shortest path we can use a standard algorithm such as Dijkstra's algorithm (Figure 134(b) gives an example). Since all the fringe nodes in the algorithm will have one of two priorities (the distance to the current vertex, or one more) the algorithm can be optimized to run in constant time for each visited vertex. If we searched the whole graph the running time would therefore be $O(nm)$, which is the same as our previous algorithms for edit distance. As we will show, however, we can get much better bounds on the number of vertices visited when the edit distance is small.

Theorem 3 $D_{ij} \geq |j - i|$

Proof: All edges in the graph have weight ≥ 0 and all horizontal and vertical edges have weight 1. Since $|j - i|$ is the minimum distance in edges from the diagonal, and the path starts on the diagonal (at 0,0), any path to location (i, j) must cross at least $|j - i|$ horizontal and vertical edges. The weight of the path must therefore be at least $|j - i|$. □

Given that Dijkstra's algorithm visits vertices in the order of their distance D_{ij}, this theorem implies that the algorithm will only search vertices that are within D_{nm} of the diagonal (see Figure 135). This leads to a running time of

$$T = O(D_{nm} \min(n, m))$$

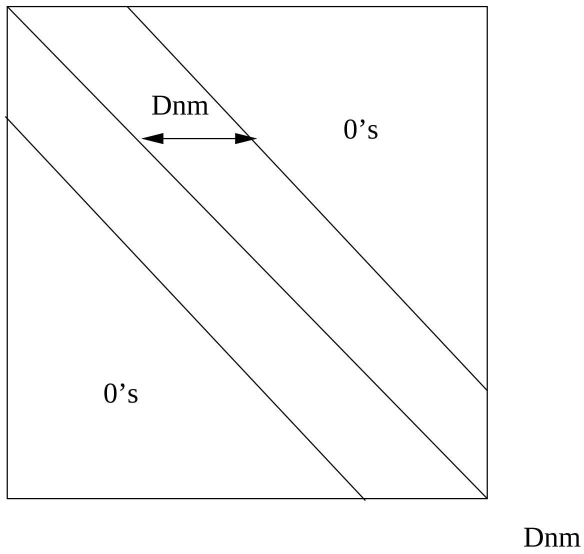


Figure 135: Bounding visited vertices

In practice programs such as `diff` do not use Dijkstra's algorithm but instead they use a method in which they try increasingly large bands around the diagonal of the matrix (see Figure 136). On each iteration the width of the band is doubled and the algorithm only fills in the entries in the band. This is continued while $D_{nm} > w/2$ where w is the width of the band. The code for filling each band is a simple nested loop.

The technique can be used in conjunction with our previous divide-and-conquer algorithm to generate a space-efficient version.

2 Introduction to Indexing and Searching

Indexing addresses the issue of how information from a collection of documents should be organized so that queries can be resolved efficiently and relevant portions of the data extracted quickly. We will cover a variety of indexing methods. To be as general as possible, a *document collection* or *document database* can be treated as a set of separate documents, each described by a set of *representative terms*, or simply *terms* (each term might have additional information, such as its location within the document). An index must be capable of identifying all documents that contain combinations of specified terms, or that are in some other way judged to be relevant to the set of query terms. The process of identifying the documents based on the terms is called a *search* or *query* of the index. Figure 137 illustrates the definitions.

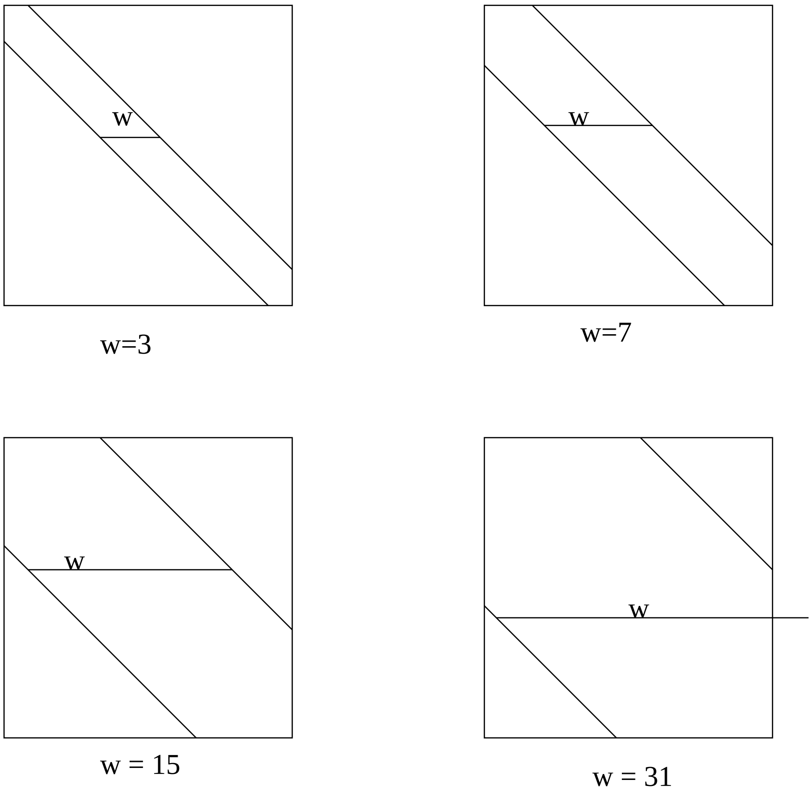


Figure 136: Using increasingly wide bands.

Applications of indexing:

Indexing has been used for many years in a wide variety of applications. It has gained particular recent interest in the area of web searching (e.g. AltaVista, Hotbot, Lycos, Excite, ...). Some applications include

- Web searches
- Library article and catalog searches
- Law, patent searches
- Information filtering, e.g. get interesting New York Time articles.

The goals of these applications:

- Speed – want minimal information retrieval latency
- Space – storing the document and indexing information with minimal space
- Accuracy – returns the "right" set of documents
- Updates – ability to modify index on the fly (only required by some applications)

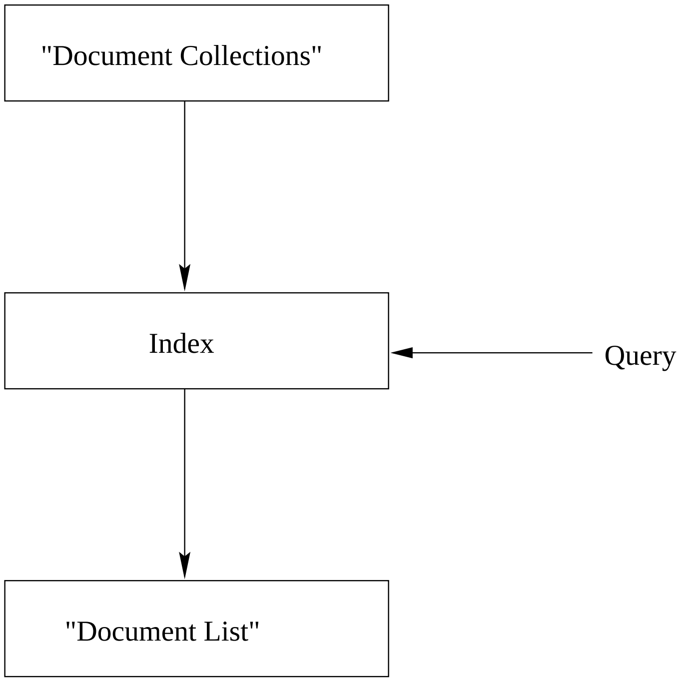


Figure 137: Overview of Indexing and Searching

The main approaches:

- Full text scanning (e.g. grep, egrep)
- Inverted file indexing (most web search engines)
- Signature files
- Vector space model

Types of queries:

- boolean (*and, or, not*)
- proximity (*adjacent, within*)
- key word set
- in relation to other documents (relevance feedback)

Allowing for:

- prefix matches (AltaVista does this)
- wildcards
- edit distance bounds (egrep)

Techniques used across methods

- case folding: London = london
- stemming: *compress* = *compression* = *compressed*
  (several off-the-shelf English language stemmers are available)
- ignore stop words: *to, the, it, be, or, ...*
  Problems arise when search on *To be or not to be* or the month of *May*
- Thesaurus: *fast* = *rapid*
  (handbuilt clustering)

Granularity of Index

The *Granularity* of the index refers to the resolution to which term locations are recorded within each document. This might be at the document level, at the sentence level or exact locations. For proximity searches, the index must know exact (or near exact) locations.

3 Inverted File Indexing

Inverted file indices are probably the most common method used for indexing documents. Figure 138 shows the structure of an inverted file index. It consists first of a *lexicon* with one entry for every term that appears in any document. We will discuss later how the lexicon can be organized. For each item in the lexicon the inverted file index has an *inverted file entry* (or *posting list*) that stores a list of pointers (also called *postings*) to all occurrences of the term in the main text. Thus to find the documents with a given term we need only look for the term in the lexicon and then grab its posting list. Boolean queries involving more than one term can be answered by taking the intersection (conjunction) or union (disjunction) of the corresponding posting lists.

We will consider the following important issues in implementing inverted file indices.

- How to minimize the space taken by the posting lists.
- How to access the lexicon efficiently and allow for prefix and wildcard queries.
- How to take the union and intersection of posting lists efficiently.

3.1 Inverted File Compression

The total size of the posting lists can be as large as the document data itself. In fact, if the granularity of the posting lists is such that each pointer points to the exact location of the term in the document, then we can in effect recreate the original documents from the lexicon and posting lists (*i.e.*, it contains the same information). By compressing the posting lists we can both reduce the total storage required by the index, and at the same time potentially reduce access time since fewer disk accesses will be required and/or the compressed lists can

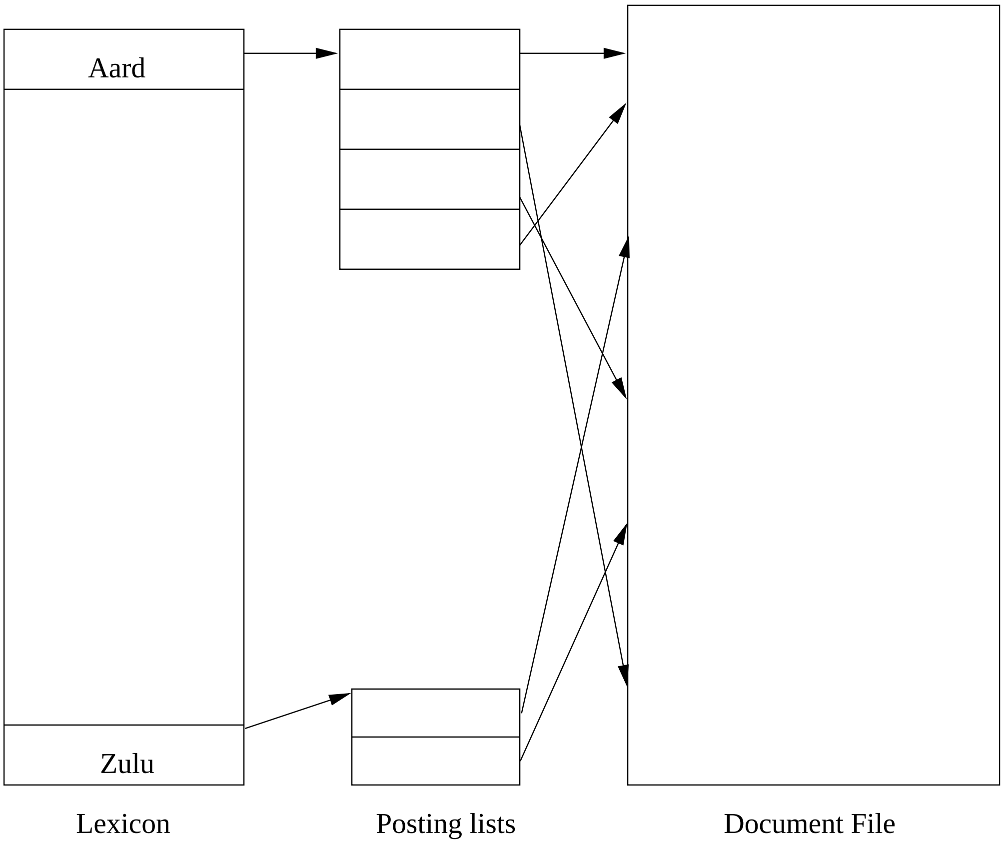


Figure 138: Structure of Inverted Index

fit in faster memory. This has to be balanced with the fact that any compression of the lists is going to require on-the-fly uncompression, which might increase access times. In this section we discuss compression techniques which are quite cheap to uncompress on-the-fly.

The key to compression is the observation that each posting list is an ascending sequence of integers (assume each document is indexed by an integer). The list can therefore be represented by a initial position followed by a list of gaps or deltas between adjacent locations. For example:

original posting list: elephant: [3, 5, 20, 21, 23, 76, 77, 78]

posting list with deltas: elephant: [3, 2, 15, 1, 2, 53, 1, 1]

The advantage of using the deltas is that they can usually be compressed much better than indices themselves since their entropy is lower. To implement the compression on the deltas we need some model describing the probabilities of the deltas. Based on these probabilities we can use a standard Huffman or Arithmetic coding to code the deltas in each posting list. Models for the probabilities can be divided into global or local models (whether the same probabilities are given to all lists or not) and into fixed or dynamic (whether the probabilities are fixed independent of the data or whether they change based on the data).

An example of a fixed model is the γ code. Think of a code in terms of the implied probability distribution $P(c) = 2^{-l(c)}$. This is the inverse of the definition of entropy. For

Decimal	γ Code
1	0
2	100
3	101
4	11000
5	11001
6	11010
7	11011
8	1110000

Table 13: γ coding for 1 through 8

example, binary code gives a uniform distribution since all codes are the same length, while the unary codes ($1 = 0, 2 = 10, 3 = 110, 4 = 1110, \ldots$) gives an exponential decay distribution ($p(1) = 1/2, p(2) = 1/4, p(3) = 1/8, p(4) = 1/16, \ldots$). The γ code is in between these two extreme probability distributions. It represents the integer i as a unary code for $1+\lfloor(\log_2(i))\rfloor$ followed by the binary code for $i - 2^{\lfloor\log_2(i)\rfloor}$. The unary part specifies the location of the most significant non-zero bit of i, and then the binary part codes the remaining less significant bits. Figure 139 illustrates viewing the γ codes as a prefix tree, and Table 13 shows the codes for 1-8. The length of the γ codes is

$$l_i = 1 + 2\log(i)$$

The implied probabilities are therefore

$$P[i] = 2^{-l(i)} = 2^{1+2\log(i)} = \frac{1}{2i^2}$$

This gives a reasonable model of the deltas in posting lists and for the TREC database (discussed in the next class) gives a factor of 3 compression compared with binary codes (see Table 14). By adjusting the code based on the length of the posting list (i.e. using a local method), the compression can be improved slightly. An example is the Bernoulli distribution (see Table 14).

In dynamic methods the probabilities are based on statistics from the actual deltas rather than on a fixed model such as the γ code. As with the static models, these methods can either be global or local. For a global method we simply measure the probability of every delta and code based on these probabilities. For local methods we could separately measure the probabilities for each posting list, but this would take too much space to store the probabilities (we would have to store separate probabilities for every list). A solution is to batch the posting lists into groups based on their length. In particular create one batch for each integer value of $\lfloor\log(length)\rfloor$. In this way we only have $\log(n)$ sets of probabilities, where n is the length of the longest posting list. Results comparing the methods are shown in Table 14 (from *Managing Gigabytes*, by Witten, Moffat, and Bell, Van Nostrand Reinhold, 1994).

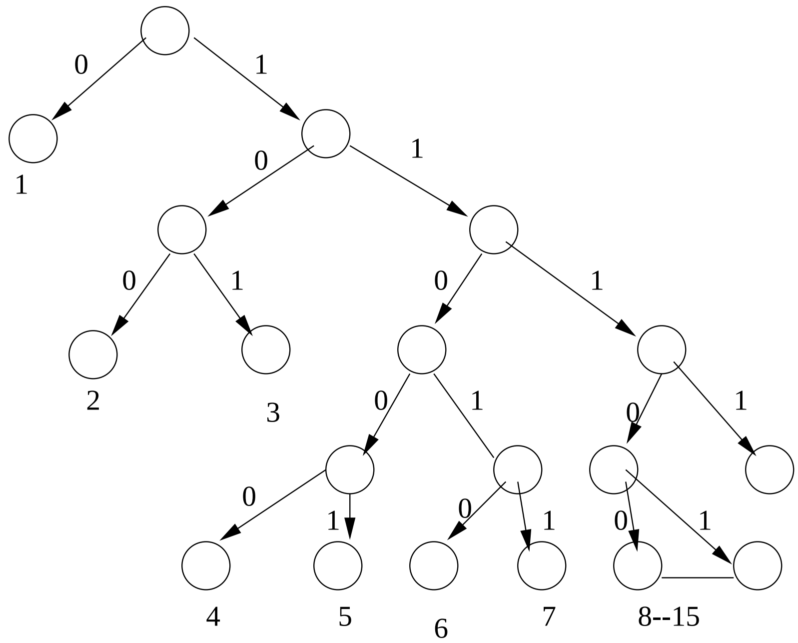


Figure 139: γ code

Method	Bits/pointer
unary	1719
binary	20
γ	6.43
Bernoulli	5.73
dynamic global	5.83
batched dynamic local	5.27

Table 14: Compression of posting files for the TREC database using various techniques in terms of average number of bits per pointer.

3.2 Representing and Accessing Lexicons

There are many ways to store the lexicon. Here we list some of them

- Sorted – just store the terms one after the other in a sorted array
- Tries – store terms as a trie data structure
- B-trees – well suited for disk storage
- Perfect hashing – assuming lexicon is fixed, a perfect hash can be calculated
- Front-coding – stores terms sorted but does not repeat front part of terms (see Table 15). Requires much less space than a simple sorted array.

Term	Complete front coding	Partial 3-in-4 front coding
7, jezebel	3, 4, ebel	0, 7, jezebel
5, jezer	4, l, r	4, l, r
7, jezerit	5, 2, it	5, 2, it
6, jeziah	3, 3, iah	3, 3, iah
6, jeziel	4, 2, el	0, 6, jeziel
7, jezliah	3, 4, liah	3, 4, liah
6, jezoar	3, 3, oar	3, 3, oar
9, jezrahiah	3, 6, rahiah	3, 6, rahiah
7, jezreel	4, 3, eel	0, 7, jezreel
11,jezreelites	7, 4, ites	7, 4, ites
6, jibsam	1, 5, ibsam	1, 5, ibsam
7, jidlaph	2, 5, dlaph	2, 5, dlaph

Table 15: Front coding (the term before jezebel was jezaniah). The pair of numbers represent where a change starts and the number of new letters. In the 3-in-4 coding every 4th term is coded completely making it easier to search for terms.

Number	Term
1	abhor
2	bear
3	laaber
4	labor
5	laborator
6	labour
7	lavacaber
8	slab

Table 16: Basic terms

When choosing among the methods one needs to consider both the space taken by the data structure and the access time. Another consideration is whether the structure allows for easy prefix queries (*e.g.*, all terms that start with `wux`). Of the above methods all except for perfect hashing allow for easy prefix searching since terms with the same prefix will appear adjacently in the structure.

Wildcard queries (*e.g.*, `w*x`) can be handled in two ways. One way is to use n-grams, by which fragments of the terms are indexed (adding a level of indirection) as shown in Table 17 for the terms listed in Table 16. Another way is to use a rotated lexicon as shown in Table 18.

Digram	Term numbers
$a	1
$b	2
$l	3,4,5,6,7
$s	8
aa	3
ab	1,3,45,6,7,8
bo	4,5,6
la	3,4,5,6,7,8
or	1,4,5
ou	6
ra	5
ry	5
r$	1,2,3, 4,5,6,7
sl	8

Table 17: N-Gram

Rotated Form	Address
\$abhor	(1,0)
\$bear	(2,0)
\$laaber	(3,0)
\$labor	(4,0)
\$laborator	(5,0)
\$labour	(6,0)
\$lavacaber	(7,0)
\$slab	(8,0)
aaber\$l	(3,2)
abhor\$	(1,1)
aber\$la	(3,3)
abor\$l	(4,2)
aborator\$l	(5,2)
abour\$l	(6,2)
aber\$lavac	(7,6)
ab\$sl	(8,3)
r\$abho	(1,5)
r\$bea	(2,4)
r\$laabe	(3,6)
r\$labo	(4,5)
r\$laborato	(5,9)
r\$labour	(6,6)
r\$lavacabe	(7,9)
slab\$	(8,1)

Table 18: Rotated lexicon