Manuscript to appear in *IEEE Transactions on Speech and Audio Processing*, Final Version, March 30, 1998.

Computational Complexity of a Fast Viterbi Decoding Algorithm for Stochastic Letter-Phoneme Transduction

R.W.P. Luk[†], *Member, IEEE*, and R.I. Damper [‡*], *Senior Member, IEEE*

[†]Department of Computing,
Hong Kong Polytechnic University,
Hung Hom,
Kowloon,
Hong Kong.
(email: `csrluk@comp.polyu.edu.hk`)

[‡]Image, Speech and Intelligent Systems (ISIS) Research Group,
Department of Electronics and Computer Science,
University of Southampton,
Southampton SO17 1BJ,
UK.
(email: `rid@ecs.soton.ac.uk`)

[*]Department of Computer Science and Engineering,
Oregon Graduate Institute of Science and Technology,
PO Box 91000
Portland
OR 97291-1000
USA.
(email: `damper@cse.ogi.edu`)

Address for correspondence:

Until 15 September 1997:

R.I. Damper, PhD,
Visiting Research Professor,
Department of Computer Science and Engineering,
Oregon Graduate Institute of Science and Technology,
PO Box 91000
Portland
OR 97291-1000
USA.

Tel: +1 503 690-1151
FAX: +1 503 690-1548
Email: `damper@cse.ogi.edu` or
`rid@ecs.soton.ac.uk` (will auto-forward)

After 15 September 1997:

Dr. R.I. Damper,
Image, Speech and Intelligent Systems (ISIS) Research Group,
Department of Electronics and Computer Science,
University of Southampton,
Southampton SO17 1BJ,
UK.

Tel: +44 1703 594577
FAX: +44 1703 594498
Email: `rid@ecs.soton.ac.uk`

Abstract

This paper describes a modification to, and a fast implementation of, the Viterbi algorithm for use in stochastic letter-to-phoneme conversion. A straightforward (but unrealistic) implementation of the Viterbi algorithm has a linear time complexity with respect to the length of the letter string, but quadratic complexity if we additionally consider the number of letter-to-phoneme correspondences to be a variable determining the problem size. Since the number of correspondences can be large, processing time is long. If the correspondences are pre-compiled to a deterministic finite-state automaton to simplify the process of matching to determine state survivors, execution time is reduced by a large multiplicative factor. Speed-up is inferred indirectly since the straightforward implementation of Viterbi decoding is too slow for practical comparison, and ranges between about 200 and 4000 depending upon the number of letters processed and the particular correspondences employed in the transduction. Space complexity is increased linearly with respect to the number of states of the automaton. This work has implications for fast, efficient implementation of a variety of speech and language engineering systems.

Keywords: *letter-to-phoneme conversion, stochastic transduction, Viterbi algorithm, finite-state automata.*

EDICS Category: *SA 1.3 Speech Synthesis; SA 1.3.1 Text Analysis and Text-to-Phoneme Translation.*

List of Figures

List of Tables

I. Introduction

Many automatic techniques for translating English word spellings to their phonemic equivalents (the word's pronunciation) have been proposed [1–4]. Stochastic transduction, introduced in [5, 6], has recently [7] demonstrated performance of about 72% and 93% word and phoneme accuracy respectively. According to the evaluations reported in [8, 9], this is far superior to traditional manually-derived rules such those described in [10–12]. The basic idea of stochastic transduction is to enumerate all possible equivalent phonemic strings of the input word spelling using a set of letter-to-phoneme correspondences, Σ. The input string is parsed according to the letter-parts of the correspondences, and the 'best' pronunciation is found as the maximum likelihood (ML) sequence of the associated phoneme-parts given some estimate of the transition probabilities of the correspondences.

However, in a brute-force implementation, the number of possible pronunciations of each word grows exponentially with the length of the input string. In a practical application such as a text-to-speech system, the number of input words to be translated is indeterminate and may be very large. Likewise, the number of possible alignments between the spelling, α_i, of a word i and its pronunciation, β_i, grows in a factorial manner with string lengths $|\alpha_i|$ and $|\beta_i|$ using elementary alignment operations. Again, this is important because the transition probability estimates for the correspondences generally have to be found (by re-estimation) from the maximum likelihood alignments for *all* words in some training corpus; furthermore, the correspondences themselves may be automatically inferred from these alignments as in [7]. Thus, we seek efficient mechanisms to find the ML alignment and translation.

Maximum likelihood string translation is a particular instance of a classical and well-studied search problem, frequently encountered in speech and language processing. Hence, a variety of techniques is available for its solution. An extended version of the stack-decoding algorithm introduced by Jelinek [13] can find the ML translation in stochastic letter-to-phoneme transduction [1]. However, the processing time of the basic algorithm can still grow exponentially with respect to $|\alpha_i|$ if the heuristic evaluation function is not well chosen [14]: it can be speeded up but at the expense of losing accuracy in finding the ML translation [15]. Damper and his colleagues [16, 17] have used path algebra for letter-to-phoneme transduction. In this case, the time complexity grows with cubic order [18] with respect to $|\alpha_i|$ and is $O(N^6)$ if we consider the number of possible phonemic equivalents for a letter substring to be a variable determining the problem size. While a beam search can be used with these techniques to speed up the translation process, this is at the risk of significant degradation in performance. An alternative to these approaches – which forms the focus of this paper – is to extend the classical Viterbi decoding algorithm [19, 20] for stochastic letter-to-phoneme transduction. This algorithm achieves its intrinsic efficiency by applying the dynamic programming principle to find the ML path through a matrix – or trellis – of probabilities.

The remainder of this paper is structured as follows. Section II describes a version of the Viterbi algorithm suited to the determination of the ML translation in stochastic letter-to-phoneme conversion. The computational complexity of the algorithm is analyzed, and extensions to allow ML alignment are also detailed. The analysis shows that the complexity of translation and alignment are dominated by the operations of matching the correspondences to the input to determine the state survivors (i.e. those states that have to be considered as possible contributors to the optimal path) and then computing the locally-optimal links for these survivors. Next, section III describes a fast implementation for translation based on the construction of a finite-state automaton to perform the matching, and pass only the relevant state survivors on to the Viterbi algorithm. Several theorems having an important bearing on the size of the required automaton are reviewed. We also show that alignment can be effected using a similar, but smaller, automaton. Section IV is concerned with the obtained speed-up, which is inferred indirectly because the straightforward Viterbi algorithm is too slow for practical comparison with the fast implementation. A measurement model is derived and shown to produce an excellent fit to obtained CPU times. This model predicts a speed-up by a large multiplicative factor of the order of 200–4000 times, depending upon the precise statistical assumptions, the set of correspondences employed, and the size of the data set to be processed. The speed-up is at the expense of additional space complexity, as described in section V. Finally, section VI concludes.

II. Modification to Viterbi Decoding

In a stochastic finite-state model of transduction as used here, states of the model can be associated with states of the trellis and Viterbi decoding applied to find the maximum likelihood state sequence (i.e. the ML translation). However, some modifications to the basic decoding algorithm are required. There are two ways to extend Viterbi decoding to suit it for letter-to-phoneme transduction: changing either the statistical assumption or the structure of the trellis. For the former, an Nth-order Markov model can implicitly represent a set of correspondences using the N-gram transition probabilities [3]. However, in a direct implementation, the number of state survivors and the number of possible state transitions both grow exponentially with N [21]. Further, the actual correspondences only form a small subset of all the possible N-grams, wasting storage space. Finally, the larger N is, the more training data are needed to estimate reliably the transition probabilities.

The alternative adopted here is to consider each correspondence as an atomic unit and then build the trellis for the input string accordingly (see [7]). That is, a state of the trellis denotes a correspondence $R_y = (\delta_y, \mu_y)$ matched at a particular position, where δ_y is the letter substring and μ_y is the corresponding phoneme substring. In this case, the first-order Markov assumption is still used and the number of survivors is simply $\psi = |\Sigma| + 1$ in the worst case (see below), where $|\Sigma|$ is the number of correspondences. Further, a much larger proportion of the possible state transitions is actually used for translation without the wastage associated with the implicit modeling of correspondences as above. Implemented as a look-up table, the storage space needed for the state-transition probabilities is ψ^2.

The modification described here is based on, and extends, our earlier work reported in [22]. It allows for state transitions not only between states s_j and s_{j-1} but between s_j and s_{j-k} also, where k is a positive integer. Possible state sequences can be represented as paths through a two-dimensional table with entries $T(x, y)$ that store intermediate results to save processing time and space. Here, x indexes position in the input string and y is an index to denote a particular correspondence R_y matched at position x, so that:

$$\begin{aligned} 0 \le x &\le |\alpha_i| + 1 \\ 0 \le y &\le |\Sigma| \end{aligned} \tag{1}$$

In the analysis which follows, we will assume that the table is implemented as an array.

The algorithm works from left to right through the input string (i.e. increasing x). At each symbol position of α_i, the letter substrings of all the correspondences R_y are matched in the right-to-left direction (i.e. backwards) from the current position x. For any matched correspondence at x, the cumulative path value is found by recursion. Assuming a first-order Markov model, the relevant recursive equation is:

$$T(x, y) = \max_t \left[T(s, t) + \log p(R_y | R_t) \right] \quad \text{with } T(0, 0) = 0 \tag{2}$$

Here, t denotes the index of any matched correspondence at position $s = x - |\delta_y|$, and (x, y) and (s, t) are bounded as in (1). A link is made between state (x, y) and the state (s, t) that gives the maximum value of $T(x, y)$. These links can either be stored along with the table entry $T(.,.)$, separately in another table, or otherwise. When the iteration has terminated, the ML translation is found from the most likely state sequence, by tracing the links from right to left along α_i starting from $(|\alpha_i| + 1, 0)$.

*** FIGURE 1 ABOUT HERE ***

This process is illustrated in Figure 1, which shows the form of the constructed table for the word *make* using the correspondences:

$$\begin{array}{lll} R_1 : (m, /\mathrm{m}/) & R_4 : (m, \epsilon) & R_7 : (a, \epsilon) \\ R_2 : (a, /\text{eɪ}/) & R_5 : (e, \epsilon) & R_8 : (a, /\text{ə}/) \\ R_3 : (k, /\mathrm{k}/) & R_6 : (ke, /\mathrm{k}/) & R_9 : (a, /\text{æ}/) \end{array} \tag{3}$$

where ϵ is the null phoneme string. The special correspondence R_0: (#, /) is reserved for word boundaries. (Note that the orthographic word delimiter # is not counted in $|\alpha_i|$.) In this example, the ML translation is /m.eɪ.k/ (where '.' denotes concatenation) corresponding to the (left-to-right) state sequence $\{R_0, R_1, R_2, R_6, R_0\}$.

A. Computational Complexity

The following discussion only identifies the major iterations where the problem size is dependent on the number of correspondences and the length of the input string. Additive constants are not included, but multiplicative constants are.

The algorithm iterates for every position x along α_i. At each of these $|\alpha_i| + 2$ positions (except the first), we suppose that it is necessary in the worst case to examine for matches all the $\psi = |\Sigma| + 1$ correspondences. In essence, the process is one of matching with each δ_y in the *backwards* direction the letter substring in the input ending with the letter at x. So, the number of iterations for matching grows in the worst case as:

$$I_M \sim |\alpha_i| \psi L \tag{4}$$

where L is the length of the longest letter substring in Σ.

For each successful such match, we then consider possible links to the previously-successful matches in column $(x - |\delta_y|)$ of the table (Figure 1) before computing the locally-optimal move to (x, y) which is then stored as the actual link for that cell. Hence, for each cell of the table, we have to examine all correspondences, and the number of iterations, I_L, for computing the links grows in the worst case as:

$$I_L \sim |\alpha_i|\psi^2 \tag{5}$$

In addition to the matching of correspondences, the algorithm has to clear all the table entries $T(.,.)$ for the next translation. Hence, the number of iterations for clearing, I_C, is quadratic in the worst case:

$$I_C \sim |\alpha_i|\psi \tag{6}$$

So, combining equations (4), (5) and (6), the total CPU time is estimated as:

$$\begin{aligned} T &\sim C_M I_M + C_L I_L + C_C I_C \\ &\sim C_1|\alpha_i|\psi^2 + C_2|\alpha_i|\psi \end{aligned} \tag{7}$$

where the C's are appropriate constants and L has been absorbed into C_2.

Since the table used to store the entries $T(.,.)$ has dimensions $|\alpha_i| + 2$ and ψ, and the number of possible state-transition probabilities is ψ^2, the required storage space S in the worst case is quadratic:

$$S \sim C_3|\alpha_i|\psi + \psi^2$$

where C_3 is a constant whose size depends upon the storage requirements for the table entries $T(.,.)$, which are cumulative log probabilities, possibly also including the locally-optimal link.

B. Alignment Extension

When re-estimating the probabilities, it is necessary to find the ML alignment [7] which is the same as for ML translation. The only difference is that the phonemic equivalent of the letter string is known, so that the input is not simply a letter string, but an array $I(.,.)$ of size $(|\alpha_i| + 2) \times (|\beta_i| + 2)$.

*** FIGURE 2 ABOUT HERE ***

A three-dimensional table with entries $A(x, y, z)$ is used to store intermediate results (Figure 2), where x and y denote a particular position in the input array $I(.,.)$ and z denotes a particular correspondence R_z matched at $I(x, y)$ in the right-to-left and bottom-to-top direction (i.e. backwards). The algorithm starts with $x = 0$, and increments y from 0 to $|\beta_i|$ for every position x. The ML alignment for R_z at (x, y) is defined (for the Markov model) according to the recursive equation:

$$A(x, y, z) = \max_t \left[A(r, s, t) + \log p(R_z | R_t) \right] \qquad \begin{cases} 0 \leq x \leq |\alpha_i| + 1 \\ 0 \leq y \leq |\beta_i| + 1 \\ 0 \leq z \leq |\Sigma| \\ A(0, 0, 0) = 0 \end{cases}$$

where t is an index of a particular correspondence, $r = x - |\delta_z|$ and $s = y - |\mu_z|$. A link is made between (x, y, z) and that (r, s, t) which yields the ML path value at $A(x, y, z)$. The ML alignment of the word is found by tracing the links back from $A(|\alpha_i| + 1, |\beta_i| + 1, 0)$.

III. Fast Implementation

Clearly, complexity is reduced if the number of state survivors is reduced below ψ: we concentrate on this aspect here. Since our transduction formalism is regular (and stochastic), it has a representation as an abstract (stochastic) finite-state automaton (FSA). Because the matching process is deterministic, however, we do not need a stochastic machine to do this. Instead, we use a deterministic FSA (DFA), which passes the matched correspondences on to the Viterbi algorithm. This then finds the locally-optimal transition to these states and, finally, the ML translation or alignment.

Our implementation exploits techniques from automata theory [23, 24] which, although more or less standard in other fields of software engineering (compiler design, information retrieval, etc.), have not previously been used in text-to-phoneme translation.

A. FSA Construction

It is not strictly necessary to examine all the correspondences and match them (in the backward direction) at all positions in the input string or array, as would be done in a straightforward implementation of the Viterbi algorithm. Instead, we can pre-compile the correspondences into the state-transition function $\delta_F(\cdot)$ of an FSA to speed up the matching process. (The reader is warned against confusing the state-transition function, δ, with the letter substring δ. Since both uses are established in the literature, we have refrained from changing one or other symbols.)

The first step is to build a trie [25, 26], augmented with transitions on the empty input ϵ, to encode the set of correspondences. Figure 3 illustrates this in simplified form for the example correspondences in (3). The trie is then used to generate the state-transition function $\delta_F(\cdot)$.

*** FIGURE 3 ABOUT HERE ***

The root of the trie is the start node S and each arc has an orthographic symbol associated with it. A letter substring δ_y of a correspondence R_y is a path from S to some other node P in the trie. Node P has an ϵ-transition to S because, after successfully matching a specific letter substring of a correspondence, the FSA must also start matching from the current position for the rest of the input string. Node P also has attached a list of correspondences having the same matched letter substring (i.e. the same path from S to P). So, when there is a state transition to P, the correspondences attached to P are exactly the nodes of the trellis in the Viterbi algorithm at that particular position in the input string. Hence, matching is effected by the simple and fast expedient of making transitions in the FSA. Further, the only states ever considered are precisely the ones that match the input. (Note too that while the FSA consumes only a single input symbol at each state transition, this is not true of the overall transduction system which includes the Viterbi decoding as a component.)

Clearly, the trie has exactly the structure of the state-transition diagram of a non-deterministic FSA (NFA). Self-transitions from and to S are implicit for all input symbols that do not lead to other nodes. Also, there are no ϵ-cycles in the trie, since all ϵ-transitions only point back to S (there are no ϵ-transitions from S).

The ϵ-transitions can be eliminated and the trie used to generate the transition function of the corresponding deterministic FSA (DFA) using the standard technique of subset construction [24, pp. 117–121]. In principle, this technique for NFA-to-DFA conversion suffers from the potential problem that the number of states in the DFA, $|Q|$, can grow exponentially in the worst case with respect to the number of states of the NFA, τ. However, this is only likely in pathological cases (e.g. for regular expressions including many repetitions of the same substring) and should not occur for exact matching of strings consisting of a few, different symbols. Thus, Aho *et al* [24] state that the exponential bound is not usually approached in practice. Indeed, by appealing to the special properties of the trie, the upper bound on $|Q|$ can be significantly reduced, in light of the following theorems (proved in [22]).

Theorem 1 *A state* s_k *of the DFA consists of a set of nodes of the trie each of which is at a different depth.*

Corollary 1 *If only the leaves of the trie have* ϵ*-transitions, then* $|Q| = \tau$.

Theorem 2 *$|Q|$ is bounded by $\prod_{i=0}^{d}(\chi_i + 1) << 2^{\tau}$ where d is the maximum depth of the trie and χ_i is the number of nodes in the trie at depth i.*

Theorem 3 *If all the nodes in the trie have ϵ-transitions leading to S, then $|Q| = \tau$.*

Theorem 3 (which is actually implicit in the Aho-Corasick construction [26] of a DFA from an NFA) does not, of course, state that $|Q| = \tau$ in all practical cases of letter-to-phoneme translation. However, we now show that this equality holds quite generally.

Let T_1 be the trie with (a) a root that has valid out-transitions for each symbol in the input alphabet and (b) an ϵ-transition back to the root for each node at depth 1. Further, let T_2 be the same trie as T_1 with the additional property that an ϵ-transition leads back to the root for every node of T_2.

Claim: The DFA transition tables of T_2 and T_1 built by subset construction are identical so that, by theorem 3, $|Q| = \tau$ for the DFA corresponding to T_1.

Proof: For T_1 to generate the same DFA as for T_2, each node in T_1 (except S) must associate with S during subset construction. We prove that it does so by induction.

Basis: For depth 0, the condition does not apply. For depth 1, it applies trivially (by property (a) of T_1).

Induction: For arbitrary depth, $0 < d < k + 1$, the DFA state corresponding to each node m of T_1 is a superset of $\{m,\ S\}$ during subset construction.

At depth k, subset construction builds a new DFA state by making a transition from a node m with corresponding DFA state M to a successor node n at depth $k + 1$. The

DFA state N corresponding to n is:

$$N = \delta_D(M, a) \qquad \text{for some } a \text{ in the input alphabet}$$

where the subscript D of the transition function denotes the DFA.

By the inductive hypothesis:

$$M = \{m,\ S\} \cup R$$

where R is the set of remaining states in the superset, so that:

$$\begin{aligned} N &= \delta_N(m,\ a) \cup \delta_N(S,\ a) \cup \delta_N(R,\ a) \\ &= \{n\} \cup \delta_N(S,\ a) \cup \delta_N(R,\ a) \end{aligned}$$

where the subscript N of the transition function denotes the NFA. Now let:

$$\delta_N(S,\ a) = P$$

where P is a DFA state which corresponds to a node p at depth 1 of the trie. By theorem 1 and the properties of T_1, $P = \{p,\ S\}$ and:

$$\begin{aligned} N &= \{n\} \cup \{p,\ S\} \cup \delta_N(R,\ a) \\ &= \{n,\ S\} \cup W \end{aligned}$$

where $W = \{p\} \cup \delta_N(R,\ a)$. Thus, node n is associated with the root S in the corresponding DFA state N.

In practice, the conditions (a) and (b) to be satisfied for T_1 are met by any set of correspondences which includes all possible single-letter correspondences. This is essentially guaranteed by the fact that every letter has a most-probable, or ‘default’, pronunciation. Hence, $|Q| = \tau$ in all practical cases.

B. Alignment Extension

The main idea for an efficient implementation to find the ML alignment is simply to extend the efficient implementation of ML translation. We use the DFA constructed for ML translation to identify the set of correspondences matched at a particular position x along α_i. At each position x, the phoneme-parts of the correspondences attached at x are compiled to produce another DFA similar to that for ML translation. This new DFA parses β_i to output a set of matched correspondences at $I(x, y)$ for all valid values of y. The construction of the DFA is identical to that for ML translation except that all the nodes in the trie have ϵ-transitions to S. This is because we need to find any substrings in β_i that match with the correspondences and then to determine whether the matched correspondences are connected to some previous path at $I(x - |\delta_z|, y - |\mu_z|)$ but not at $I(x, y - |\mu_z|)$. According to theorem 3, $|Q| = \tau$ for this new DFA. Since there is only a small number of correspondences matched at a particular position x, the number of states of the DFA, ζ, is small (typically around 10 to 20).

IV. Inferring Speed-Up

In this work, the DFAs for translation and alignment were implemented as state-transition tables. (Note, however, that in [7] we used adjacency lists.) No attempt was made to minimize the DFAs.

The obvious way to measure the speed-up is to compare CPU times for the fast (FSA-based) and straightforward implementations. By so doing, we do not imply that the straightforward implementation is in any way a realistic contender for use in a practical system: this simplest possible implementation merely serves as a convenient baseline for comparison. To avoid questions of precisely how much knowledge of the correspondences and their structure it is reasonable to assume, we simply assume none. Attempts to measure speed-up directly in this way were, however, quickly abandoned in view of the very long execution times for the straightforward implementation (even with the smallest set of correspondences and dummy transition probabilities). Instead, we decided to predict or infer the speed-up indirectly.

A. Measurement Model

To derive a statistical predictor for the speed-up, let $f_{k,\Sigma}$ be a random variable that denotes the number of states emitted per symbol read – where k is an index of the symbol and Σ indexes the set of correspondences employed. Considering both ML translation and alignment, the total number of symbols involved in the computation (Figure 2) is:

$$n = \sum_i (|\alpha_i| + 2)(|\beta_i| + 2)$$

where i is the index of all words in the training and/or test sets. We regard translation and alignment as sampling processes of size n, from an underlying distribution of the number of states emitted per symbol. The distribution is dependent on the particular set of correspondences used, which yield different means and variances for $f_{k,\Sigma}$.

According to equation (7), the total CPU time for the straightforward implementation is a quadratic in the number (ψ) of states emitted. For the fast implementation, however, the number of state survivors is reduced such that only *relevant* states are considered. Hence, we model the CPU time for the fast implementation as a quadratic in the reduced number of states emitted. Taking account of an additive constant A:

$$T_\Sigma = C_1 \sum_k f_{k,\Sigma}^2 + C_2 \sum_k f_{k,\Sigma} + A \qquad (8)$$

where k ranges from 1 to n. We assume constants C_1 and C_2 to be essentially identical in the two cases (straightforward and fast implementation). Since n is large in practice ($\sim 10^5 - -10^7$):

$$
\begin{aligned}
\sigma_\Sigma^2 &\sim \frac{\sum_k \left(f_{k,\Sigma} - \overline{f_\Sigma}\right)^2}{n} \\
&= \frac{\sum_k f_{k,\Sigma}^2 - 2\sum_k f_{k,\Sigma}\overline{f_\Sigma} + \sum_k \overline{f_\Sigma}^2}{n} \\
&\sim \frac{\sum_k f_{k,\Sigma}^2 - \sum_k \overline{f_\Sigma}^2}{n} \qquad \text{assuming } \sum_k f_{k,\Sigma}\overline{f_\Sigma} \sim \sum_k \overline{f_\Sigma^2} \\
\text{and} \quad \sigma_\Sigma^2 &\sim \frac{\sum_k f_{k,\Sigma}^2}{n} - \overline{f_\Sigma}^2
\end{aligned}
$$

Assuming further that the standard deviation does not vary significantly across different sets of correspondences, $\sigma_\Sigma = \sigma$, then:

$$\sum_k f_{k,\Sigma}^2 = n\left(\sigma^2 + \overline{f_\Sigma}^2\right)$$

Substituting this and $n\overline{f_\Sigma} = \sum_k f_{k,\Sigma}$ into equation (8) yields:

$$T_\Sigma = C_1 n\overline{f_\Sigma}^2 + C_2 n\overline{f_\Sigma} + A' \tag{9}$$

where $A' = A + C_1 n\sigma^2$ (i.e. another additive constant).

In this work, we use the mean number of emitted states per symbol, $\overline{f_\Sigma}$, as the predictor of the speed-up according to equation (9). To examine its adequacy, we fit quadratic polynomials using regression analysis to the total CPU time spent in translation and alignment and examine the obtained correlation.

B. Results

All results were obtained using the machine-readable Oxford Advanced Learner's Dictionary [27]. Transduction computations were performed on a DEC MicroVAX II with 20 Mbyte memory. Actual performance in terms of correct translation and alignment is considered elsewhere [6, 7, 29, 30, 31]: our concern here is solely with the efficiency of the implementation.

Three statistical models were used to carry out ML translation and alignment for the approximately 20,000 words in the dictionary. These were (see [7]):

Independent: an unrealistically simple model in which each letter substring δ_y matched to a correspondence on the ML path receives as its phoneme string μ_j its most common pronunciation, independent of the previously matched correspondence R_t.

Markov: in which probabilities are conditioned on the previous state (i.e. correspondence R_t) only according to the Markov assumption, and as expressed in equation (2).

Hidden Markov: involving two independent stochastic processes, the first governing the probability of letter-substring emission in a given state, and the second governing the transition probability between states according to the Markov assumption.

This was repeated three times (i.e. about 60,000 ML translations and 60,000 alignments) to reduce the sensitivity of the results to extraneous operating-system loading conditions.

*** TABLE I ABOUT HERE ***

Table I shows various statistics and measures associated with the 8 sets of correspondences studied for the Markov statistical model. As identified in the table, these sets were:

Ains: the manually-compiled context-dependent rewrite rules of Ainsworth [10] with the context-dependency simply ignored.

Elo: the manually-compiled context-dependent rewrite rules of Elovitz *et al* [11], again with the context-dependency ignored.

LK: the manually-compiled correspondences tabulated by Lawrence and Kaye [28]. In the small number of cases where the correspondences were context-dependent, the context-dependency was again ignored.

DD: the correspondences automatically inferred by the dynamic programming and delimiting method in [29].

HMM: the set inferred on the basis of hidden Markov statistics in [30].

LK+Aff: the Lawrence and Kaye correspondences with added affixes as in [31].

DD+Aff: the DD set with added affixes as above.

GST: the correspondences automatically inferred by the generalized stochastic transducer method in [30].

Best-fit quadratics were obtained for the three statistical models by regression [32] based on equation (9), i.e. treating the CPU time as a measure of T_Σ. Note, however, that the GST correspondences give rise to an outlier (i.e. $\overline{f_\Sigma} \sim 42$ in Table I). Since the presence of this outlier could conceivably distort the outcome, the regression analysis was performed both with and without the GST correspondences included.

*** TABLE II ABOUT HERE ***

Table II shows the resulting goodness of fit as measured by the correlation coefficient r for the quadratic regression. It does appear that the outlier has influenced the results somewhat, yielding a rather higher correlation in most cases than when it is excluded. It is also clear from the excellent correlation (always > 0.94) that $\overline{f_\Sigma}$ is indeed a very good predictor of speed-up. Further, the assumptions on which equation (9) is based (viz. σ_Σ is constant for all sets Σ of correspondences, $\sum_k f_{k,\Sigma}\overline{f_\Sigma} \sim \sum_k \overline{f_\Sigma}^2$) is vindicated – at least for the sets studied here.

C. Estimating the Speed-Up Factor λ

The quadratic regression equation [32] for the Markov statistical model (with outlier) was found to be:

$$T = 0.6499\overline{f_\Sigma}^2 + 0.086\overline{f_\Sigma} + 56.2$$

with correlation coefficient 0.9995. The t-ratios were 13.64, 0.04 and 2.91 for the quadratic, linear and constant terms respectively, with corresponding p values of 0.000, 0.972 and 0.033. Its very high p value indicates that the linear term with coefficient C_2 can be disregarded, while the effectively zero value for the quadratic term indicates that it dominates. Similar results were obtained for the other statistical models (independent and hidden Markov).

*** FIGURE 4 ABOUT HERE ***

To confirm the dominance of the quadratic term, a linear regression analysis was carried out of the measured CPU time against $\overline{f_\Sigma}^2$, the square of the mean number of states emitted. Figure 4(a) and (b) depicts the results for the Markov statistical model. In the case of the GST outlier (the largest set of correspondences with the largest value of $\overline{f_\Sigma}^2$), the processing time was about 100 words per minute. This increased to approximately 1200 words per minute for the more representative LK correspondences. In (a), all the data (for all 8 sets of correspondences) are shown: in (b), the GST outlier has been excluded. The obtained correlation coefficients were 0.9991 and 0.9851 respectively – not significantly different from the results of the quadratic regression. Slopes were 0.6543 with the outlier and 0.6242 without. It is clear from this very strong correlation that the linear term with coefficient C_2 can indeed be ignored. That is, the cost of the dynamic programming optimization (to find the locally-optimal links between state survivors) dominates the time complexity. Similar findings obtained for the other statistical models (independent and hidden Markov).

However, it is not clear from its p value of 0.033 (for the Markov model) if the additive constant can also be ignored: the intercept at $\psi^2 = 0$ in Figure 4 is quite large – approximately 60 minutes. Hence, we define the measure of speed-up, λ, as the ratio of equations (7) and (9) with C_2 ignored but the additive constant(s) retained. Assuming the same number of symbols n is processed in the two cases and the additive constants are the same (i.e. $C_1 n\sigma^2 << A$), this ratio is:

$$\lambda = \frac{C_1 n\psi^2 + A}{C_1 n\overline{f_\Sigma}^2 + A}$$

Reasonable approximate values of $C_1 n$ and A can be taken from the foregoing as 0.6 and 60 respectively. The last column of Table I lists the resulting values of speed-up, λ. Of course, in light of the dependence on A, λ is somewhat sensitive to the size of the data set – but arguably it is only reasonable that it should be.

V. Added Space Complexity

The implementation described reduces processing time, but at the expense of increasing storage space. The state-transition table of the DFA for orthographic matching needs at most $|Q||\Sigma_o|$ integers where $|\Sigma_o|$ is the size of the input alphabet. The storage space needed for the DFA for phoneme strings is bounded by $|Q||\Sigma_p|\zeta$, where $|\Sigma_p|$ is the size of the output (phonemic) alphabet and ζ is the maximum number of states of the DFA for phoneme strings. The total additional space requirement for both ML translation and alignment is:

$$S_a \leq |Q|\left(|\Sigma_o| + |\Sigma_p|\zeta\right) \text{ storage locations}$$

which is linear with respect to $|Q|$ if we consider $|\Sigma_o|$, $|\Sigma_p|$ and ζ to be relatively small constants. Typically, $|\Sigma_o| \sim 30$, $|\Sigma_p| \sim 50$ and $\zeta \sim 20$. Then $S_a \leq 1030|Q|$ for ML translation and alignment, and $S_a \leq 30|Q|$ for ML translation only.

This assumes that the space required for Viterbi decoding in the fast implementation is equal to that required in the straightforward implementation – so that the only difference is the space needed for the DFAs. This is a worst-case scenario. Obviously, the reduced number of state survivors will allow some saving for the fast implementation.

VI. Conclusions

We have described a modification of the Viterbi algorithm for stochastic (maximum likelihood) letter-to-phoneme translation as an alternative to the stack-decoding algorithm and the path algebra technique. We have also described how the Viterbi algorithm finds the ML alignment. This algorithm has a linear dependence on the length of the input string together with a quadratic dependence on the number of letter-to-phoneme correspondences and a quadratic space complexity. A fast implementation for translation has been described which achieves its speed-up by building a deterministic finite-state automaton. Since there is currently no standard method for deploying finite-state technology in text-to-speech systems, speed-up is assessed relative to a simple, straightforward, and admittedly unrealistic, implementation which facilitates theoretical comparison.

The DFA performs fast matching of letter substrings and passes on to the Viterbi algorithm only those state survivors that are relevant to the dynamic-programming optimization. The speed-up is obtained at the cost of added space complexity proportional to the number of states, $|Q|$, of the DFA in the worst case. A similar (but generally smaller) DFA is used for ML alignment.

Because the straightforward Viterbi algorithm was too slow for practical comparison, the speed-up factor has been inferred indirectly. We show that the mean number of correspondences associated with each state of the DFA, $\overline{f_\Sigma}$, can yield an accurate prediction of the speed of the fast algorithm. This is verified mathematically and empirically if the variance of the number of state emissions is sensibly constant across different sets of correspondences. The inferred speed-up (λ) ranges between about 200 and 4000, depending upon the number of symbols processed and the particular correspondences employed in the transduction. Hence, a speed-up factor of several hundred-fold represents a reasonable expectation in most circumstances.

The techniques described and developed in this paper have wide applicability to the class of regular translation and alignment problems which abound in speech technology and computational linguistics.

References

[1] J.M. Lucassen and R.L. Mercer, "An information theoretic approach to the automatic determination of phonemic baseforms," *Proc. ICASSP '84, IEEE Int. Conf. Acoust. Speech Signal Process.*, San Diego, CA, pp. 42.5.1–42.5.4, 1984.

[2] T.J. Sejnowski and C.R. Rosenberg, "Parallel networks that learn to pronounce English text," *Complex Systems*, vol. 1, pp. 145–152, 1987.

[3] S.H. Parfitt and R.A. Sharman, "A bi-directional model of English pronunciation," *Proc. Eurospeech '91, 2nd European Conf. Speech Communication and Technology*, Genoa, Italy, vol. 2, pp. 801–804, 1991.

[4] R.I. Damper, "Self-learning and connectionist approaches to text-phoneme conversion," in *Connectionist Models of Language and Memory*, J. Levy, D. Bairaktaris, J. Bullinaria and P. Cairns, Eds. London: UCL Press, pp. 117–144, 1995.

[5] R.W.P. Luk and R.I. Damper, "Stochastic transduction for English text-to-phoneme conversion," *Proc. Eurospeech '91, 2nd European Conf. Speech Communication and Technology*, Genoa, Italy, vol. 2, pp. 779–782, 1991.

[6] R.W.P. Luk, *Stochastic Transduction for English Grapheme-to-Phoneme Conversion*. PhD Thesis, Department of Electronics and Computer Science, University of Southampton, UK, 1992.

[7] R.W.P. Luk and R.I. Damper, “Stochastic phonographic transduction for English,” *Computer Speech and Language*, vol. 10, pp. 133–153, 1996.

[8] R.I. Damper and J.F.G. Eastmond, “Pronunciation by analogy: Impact of implementational choices on performance,” *Language and Speech*, vol. 40, pp. 1–23, 1997.

[9] R.I. Damper and K. Gustafson, “Evaluating the pronunciation component of a text-to-speech system,” *UK Speech and Language Technology (SALT) Workshop on Evaluation in Speech and Language Technology*, Sheffield, pp. 72–79, July 1997.

[10] W.A. Ainsworth, “A system for converting English text into speech,” *IEEE Trans. Audio and Electroacoust.*, vol. AU-21, pp. 288–290, 1973.

[11] H.S. Elovitz, R. Johnson, A. McHugh and J.E. Shore, “Letter-to-sound rules for automatic translation of English text to phonetics,” *IEEE Trans. Acoust. Speech Signal Process.*, vol. ASSP-24, pp. 446–459, 1976.

[12] S. Hunnicutt, “Grapheme-to-phoneme rules: a review,” *Speech Transmission Laboratory Quarterly Progress and Status Report, Royal Institute of Technology (KTH), Stockholm*, vol. STL-QPSR 2-3/1980, pp. 38–60, 1980.

[13] F. Jelinek, “Fast sequential decoding algorithm using a stack,” *IBM J. Res. Dev.*, vol. 13, pp. 675–685, 1969.

[14] N.J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

[15] B.D. Paul, “Algorithms for an optimal A^* search and linearizing the search in the stack decoder,” *Proc. ICASSP ’91, IEEE Int. Conf. Acoust. Speech Signal Process.*, Toronto, Canada, vol. 1, pp. 693–696, 1991.

[16] S.M. Lucas and R.I. Damper, “Syntactic neural networks for bidirectional text-phonetics translation,” in *Talking Machines: Theories, Models and Applications*, G. Bailly and C. Benoît, Eds. Amsterdam: Elsevier (North-Holland), pp. 127–141, 1992.

[17] K.P.H. Sullivan and R.I. Damper, “Novel-word pronunciation: a cross-language study,” *Speech Communication*, vol. 13, pp. 441–452, 1993.

[18] S.M. Lucas, private communication.

[19] A.J. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 260–269, 1967.

[20] G.D. Forney Jr., “The Viterbi algorithm,” *Proc. IEEE*, vol. 61, pp. 268–278, 1973.

[21] N. Abramson, *Information Theory and Coding*. New York: McGraw Hill, 1963.

[22] R.W.P. Luk and R.I. Damper, R.I., “A modification of the Viterbi algorithm for stochastic phonographic transduction,” *Proc. of Int. Conf. Spoken Lang. Process. (ICSLP ’92)*, Banff, Canada, vol. 2, pp. 855–858, 1992.

[23] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata, Languages and Computation*. Reading, MA: Addison-Wesley, 1979.

[24] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[25] E. Fredkin, “Trie memory,” *Comm. ACM*, vol. 3, pp. 490–500, 1960.

[26] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," *Comm. ACM*, vol. 18, pp. 333–340, 1975.

[27] *Oxford Advanced Learner's Dictionary of Current English, 3rd Edition, Electronic Handbook*. Oxford, UK: Oxford University Press, 1989.

[28] S.G.C. Lawrence and G. Kaye, "Alignment of phonemes with their corresponding orthography," *Computer Speech and Language*, vol. 1, pp. 153–165, 1986.

[29] R.W.P. Luk and R.I. Damper, "Inference of letter-phoneme correspondences by delimiting and dynamic time warping techniques," *Proc. ICASSP '92, IEEE Int. Conf. Acoust. Speech Signal Process.*, San Francisco, CA, vol. 2, pp. II.61–II.64, 1992.

[30] R.W.P. Luk and R.I. Damper, "Inference of letter-phoneme correspondences using generalised stochastic transducers," *Proc. ICSP, IEEE Int. Conf. Signal Process.*, Beijing, China, Vol. 1, pp. 650–653, 1993.

[31] R.W.P. Luk and R.I. Damper, "Experiments with silent-*e* and affix correspondences in stochastic phonographic transduction," *Proc. Eurospeech '93, 3rd European Conf. Speech Communication and Technology*, Berlin, Germany, vol. 2, pp. 917–920, 1993.

[32] *Minitab Reference Manual: Release 9 for Windows*. State College, PA: Minitab Inc., 1993.

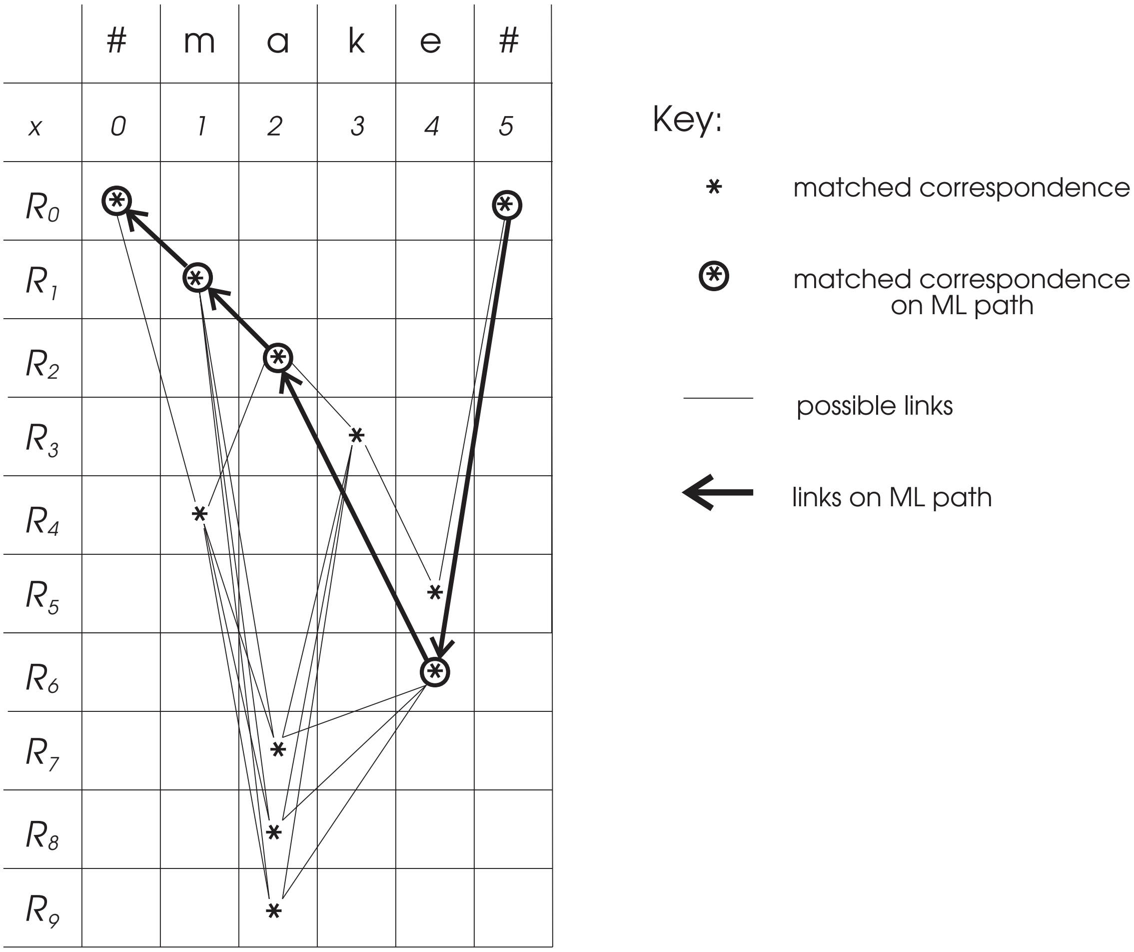


Figure 1: The two-dimensional table used to find the ML translation for the word *make* given the correspondences listed in (3). The matched correspondences are the states of the trellis, the possible links are the allowed state transitions in the trellis, and the ML path is found by dynamic programming, giving the pronunciation /m.eɪ.k /.

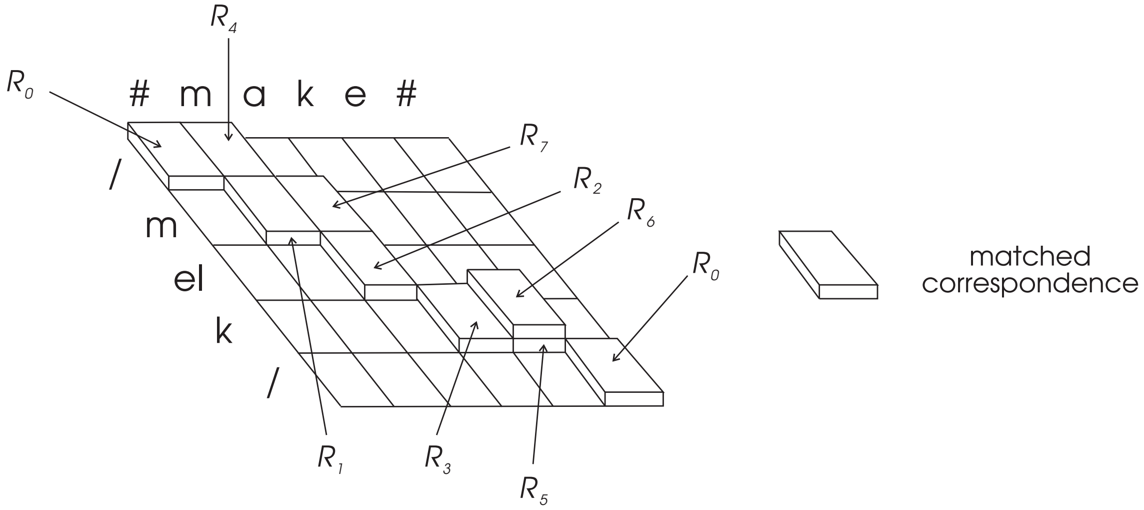


Figure 2: The ML alignment is found using the three-dimensional table illustrated here for the word (*make*, /meɪk/). Possible links are omitted for clarity.

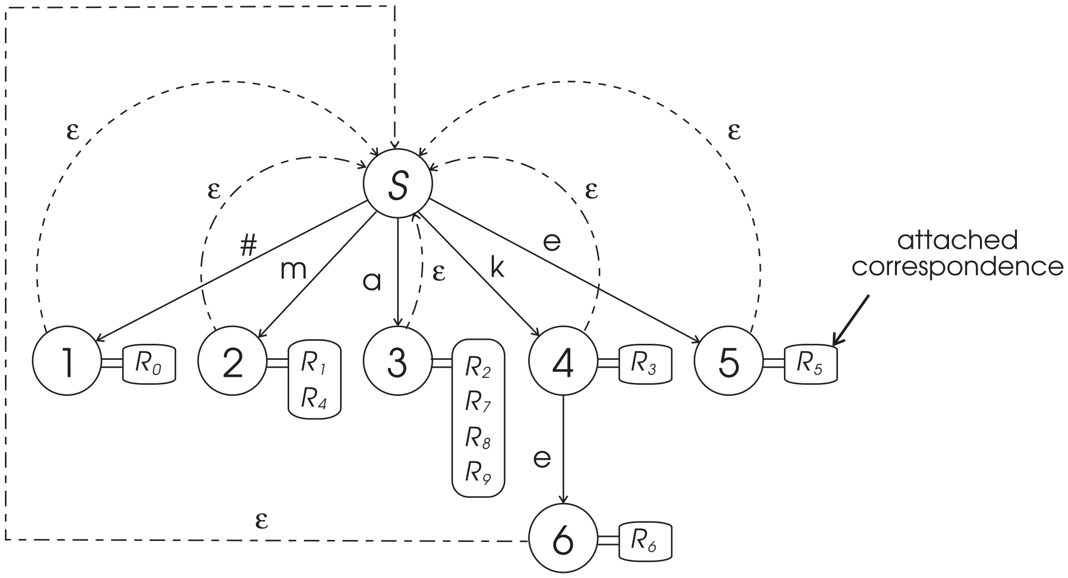
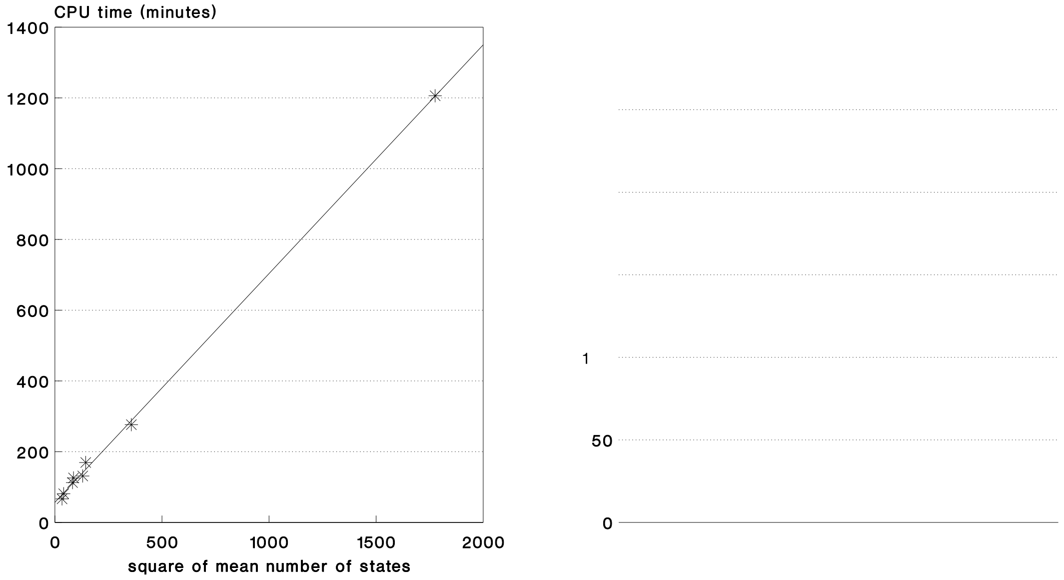


Figure 3: The trie constructed for the set of correspondences in (3). The correspondences attached to a DFA state s_k are the union of all the correspondences attached to the trie nodes that belong to s_k.


CPU time (minutes)
1400
1200
1000
800
600
400
200
0
0
500
1000
1500
2000
square of mean number of states
1
50
0

(a)