# AN OPTIMAL PATH CODING SYSTEM FOR DAWG LEXICON-HMM

*Alain Lifchitz* [*]

Laboratoire d'Informatique de Paris 6
Université P. & M. Curie & CNRS (UMR 7606)
8, rue du Capitaine Scott
75015 Paris, France
alain.lifchitz@lip6.fr

*Frederic Maire*

School of S.E.D.C.
Faculty of Information Technology
2 George Street, GPO Box 2434
Brisbane Q4001, Australia
f.maire@qut.edu.au

*Dominique Revuz*

Laboratoire d'Informatique
Institut Gaspard Monge
bât. Copernic, 5, boulevard Descartes
77454 Marne-la-vallée Cedex 2, France
dominique.revuz@univ-mlv.fr

## ABSTRACT

*Lexical constraints on the input of speech and on-line handwriting systems improve the performance of such systems. A significant gain in speed can be achieved by integrating in a digraph structure the different Hidden Markov Models (HMM) corresponding to the words of the relevant lexicon. This integration avoids redundant computations by sharing intermediate results between HMM's corresponding to different words of the lexicon. In this paper, we introduce a token passing method to perform simultaneously the computation of the a posteriori probabilities of all the words of the lexicon. The coding scheme that we introduce for the tokens is optimal in the information theory sense. The tokens use the minimum possible number of bits. Overall, we optimize simultaneously the execution speed and the memory requirement of the recognition systems.*

## 1. INTRODUCTION

A number of pattern recognition problems like hand gesture recognition, on-line and off-line *Hand Writing Recognition* (HWR) and *Automatic Speech Recognition* (ASR) can be solved by performing an elastic matching between an input pattern and a set of prototype patterns. In all these applications, the a posteriori probabilities of a number of different words are computed given a sequence of frames (feature vectors). These a posteriori probabilities are computed by running *Viterbi Algorithm* (VA) [14, 3] on the *Hidden Markov Models* (HMM) corresponding to the different words [10].

Most cursive HWR and ASR systems use a lexical constraint to help improve the recognition performance. Traditionally, the lexicon is stored in a *trie* [4]. This approach has been extended with solutions based on a more compact data structure, the *Directed Acyclic Word Graph* (DAWG) [5, 13, 6]. The non-deterministic node-automata we use to represent the lexicons can be significantly more compact than their deterministic counterparts [7]. **Figure 2** shows a non-deterministic node-automaton generating the same language as the trie of **Figure 1**.

Node-automata are better at HMM factorization because in a node-automaton the processing is done in the nodes and the routing is done with the arcs, whereas with traditional automata (that we call arc-automata), these two tasks are not separated. In a nutshell, the nodes of our automata encapsulate HMM corresponding to letters. The resulting super-structure is called a *lexicon-HMM*.

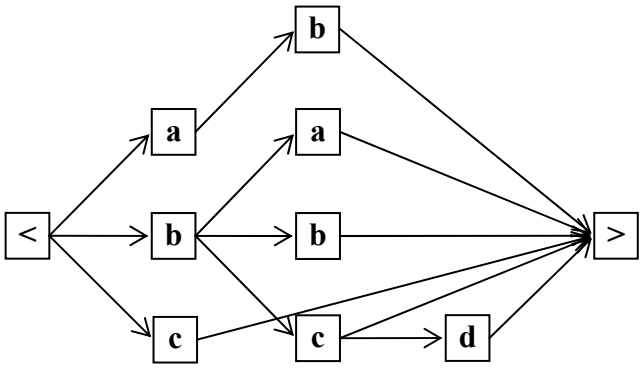


**Figure 1**. A toy example showing the node-automaton associated to a trie. This node-automaton represents the six word lexicon {'ab', 'ba', 'bb', 'bc', 'bcd', 'c'} of 12 letters. This automaton is a trie with an added common sink for all the leaves. The automaton contains 10 nodes.

The rest of the paper is structured as follows. In Section 2, we recall the basics of Viterbi algorithm, describe token passing methods, and present some methods to optimize the time and space complexity. In the same section, optimized computations of the $n$-best solutions are presented. In Section 3, we introduce a path coding system that provides an optimal token tagging scheme for Viterbi algorithm.

[*] *Correspondence to* : A. Lifchitz

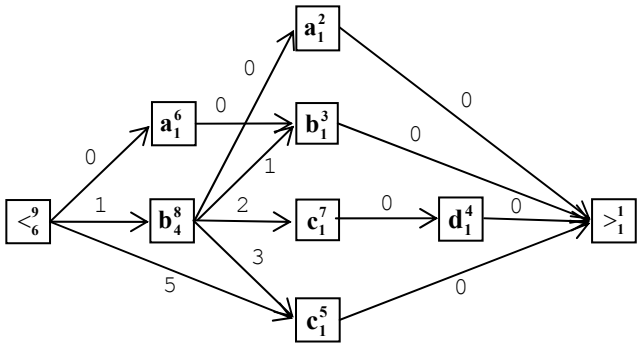


**Figure 2**. The six word toy lexicon of **Figure** 1 as DAWG node-automaton. The automaton contains 9 nodes. Nodes are labeled with a reverse topological sort index (superscript of the node letter) (see Section 2.2) and with their $\text{suff}(x)$ values (subscript of the node letter) (see Section 3.1). Arcs are labeled with their PPH increments.

## 2. VITERBI ALGORITHM

Viterbi Algorithm computes the likelihood that a given HMM generates a given string of symbols by using *Dynamic Programming* [1, 9].

As illustrated in **Figure 2**, each path in a lexicon-HMM joining the start state to a terminal state corresponds uniquely to a word of the lexicon and reciprocally.

Let $a_{ij}$ be the transition probability from state $i$ to state $j$ in the lexicon-HMM, and let $b_j(o)$ be the emission probability of symbol $o$ in state $j$. Finally, let $\delta_t(j)$ be the maximum log probability (score of the most likely sequence of hidden states) of the HMM model being in state $j$ after generating the sequence of symbols $(o_1,...,o_t)$.

The time series $\delta_t(j)$ satisfy the following recurrence relation, called the *Standard Viterbi Decoder Equation* (SVDE)

$$\delta_t(j) = \max_{i\in \text{pred}(j)} \{\delta_{t-1}(i) + \log(a_{ij})\} + \log(b_j(o_t))$$

If for each state $j$, we record its maximizing predecessor, we can easily determine the sequence of states that is the most likely to generate the sequence of observed symbols $(o_1,...,o_T)$, by first identifying which state $\hat{j}_T$ maximizes $\delta_T(j)$, then tracing back (*backtracking*) the rest of the most likely sequence of states with the sequence of maximizing predecessors starting from state $j_T$.

### 2.1. Basic token passing implementation (arbitrary sort)

As Young et al [15] pointed out, at any time step, only a single value $\delta_t(j)$, and the corresponding best *partial path* $\hat{p}_t(j)$ from the initial state to $j$, needs to be stored in state $j$. A token passing approach is best viewed as a systolic propagation of tokens over a network.

In a token passing implementation of VA, each node $j$ holds a single token $\tau_t(j)$, containing $\delta_t(j)$ and some information (called *Path History*) to represent the optimal partial path $\hat{p}_t(j)$, from the root node to $j$, via the sequence of maximizing predecessors.

**Pseudo-code 1**. Basic token passing implementation of VA

```
Initialize the token of the root.
  for t = 1:T
    for each j ∈ (j_1, j_2, ..., j_N)  // arbitrary order
      Compute τ_t(j) using the SVDE on τ_{t-1}(i).
    endfor
  endfor
```

In practice, the values of $\tau_t(j)$ and $\tau_{t-1}(j)$ are stored in two different flip-flop pointed arrays. Although the running time of this implementation of VA is optimal, some memory space is wasted.

### 2.2. Memory space optimized token passing implementation (reverse topological sort)

If we ignore the loops of the states, the HMM that we consider are acyclic. Recall that a sequence $(j_1,...,j_N)$ of states is compatible with the *Topological Sort* [12] if the sequence is an indexing of the states such that if $a < b$ then $j_a$ is not a descendant of $j_b$. The superscripts of the nodes of the toy DAWG node-automaton in **Figure** 2 come from such a sequence. A simple change in the order which the tokens are passed halves the memory requirement of VA:

**Pseudo-code 2**. Space optimized token passing implementation of VA[1]

```
// PRE: (j_1, j_2, ..., j_N) is a sequence compatible
// with the topological sort. This sequence is
// computed only once for a given HMM.
Initialize the token of the root.
  for t = 1:T
    // Scan states in reverse topological sort.
    for each j ∈ (j_N, j_{N-1}, ..., j_1)
      Compute τ_t(j) using the SVDE on τ_{t-1}(i).
    endfor
  endfor
```

Because we visit the states in reverse topological order, the same memory variable can be used to store $\tau_t(j)$ and $\tau_{t-1}(j)$. When $\tau_{t-1}(j)$ is overwritten by $\tau_t(j)$, the value of $\tau_{t-1}(j)$ is no longer needed. Whereas, if the states were scanned in the order $(j_1, j_2, \ldots, j_N)$, then $\tau(j)$ would require two distinct memory variables; one for time $t$ and one for time $t-1$. This ordering problem is a special case of the scheduling problem with precedence constraints in DAG [12]. Moreover, if node data are stored in reverse

[1] A more detailed, and improved, form of loop internal pseudo-code is presented later for the $n$-best case and can apply as well to 1-best. The computational complexity stays the same.

topological order, VA update steps can be done by simply incrementing a memory address pointer (thereby achieving optimal memory access speed).

### 2.3. $n$-best Token Passing Viterbi Algorithm

It is often desirable in practice to determine not just the best path (sequence of states or nodes depending the resolution) that maximizes the total score, but the $n$-best different paths [8]. In the context of this paper, different paths mean different words. Indeed, considering hypotheses other than the one corresponding to the best path increases the chances of finding the correct word. In many applications, some information not used in the HMM recognizer, such as a more precise grammar or a language model or other contextual clues, is used to re-rank the candidate words (improving the recognition rate).

Thanks to Bellman principle of optimality [1], it is sufficient to keep the list of the best $n$ tokens at each state in order to determine the $n$-best paths. This list is a sorted list of $n$ tokens $\boldsymbol{\tau}_t(j,n) \equiv (\tau_t(j,1),...,\tau_t(j,n))$ where $\tau_t(j,1)$ is the best token.

**Pseudo-code 3**. Naïve merging of $n$-best solutions in token passing VA

```
// Update of the n-best tokens τ_t(j,n) of state
// j at time t.
Initialize τ_t(j,n) to void.
for each i ∈ pred(j) // Scan the predecessors of j.
  for k = 1:n // if any
    On the token τ_{t-1}(i,k) use update equations
      δ_t(j) = (δ_{t-1}(i,k) + log(a_ij)) + log(b_j(o_t))
      p̂_t(j) = p̂_{t-1}(i,k) + j
    to build a candidate token τ_t(j) to be merged
    in sorted list τ_t(j,n). Keep only, if any, n
    tokens with different p̂_t(j)
  endfor
endfor
```

In the above naïve implementation of the $n$-best VA the inner loop, including merging, is executed systematically $n \times |\text{pred}(j)|$ times. So the time complexity is $n$ times the 1-best complexity plus the complexity of merging operations.

The order of execution of the loops does matter. A slightly more efficient implementation is obtained by changing the loop order as illustrated in the pseudo-code below:

**Pseudo-code 4**. Improved merging of $n$-best solutions in token passing VA

```
// Improved update of the n-best token τ_t(j,n)
// of state j at time t.
Initialize τ_t(j,n) to void.
  for k = 1:n
    for each i ∈ pred(j)
      From token τ_{t-1}(i,k) calculate
        δ_t(j) = δ_{t-1}(i,k) + log(a_ij)
      Test if δ_t(j) has to be merged in
      (τ_t(j,k),..., τ_t(j,n))
      If "merged", update the token τ_t(j) with
      p̂_t(j) = p̂_{t-1}(i,k) + j and if needed, delete
      the worst token with same p̂_t(j)
    endfor
    δ_t(j,k) = δ_t(j,k) + log(b_j(o_t)) in τ_t(j,k)
  endfor
```

Some update operations can be conditional or extracted from the most internal loop leading to significantly more efficient computations:

- Incrementation of best partial path is restricted to merged tokens.
- Final incrementation is only, and usefully, done on the $n$-best tokens for the step.
- Due to the principle of optimality [1], a simplified merging operation occurs "from the $k^{th}$ element to the end of the list" only if needed.

### 2.4. Time and space complexities

Let $N$ be the total number of characters of a lexicon-HMM: the number of states of the HMM is just few time this number. The factor is the mean number of states per character, typically in 3 in ASR and 3-7 in HWR. Let $T$ be the length of the input sequence of symbols. It is easy to see that the time complexity (theoretical worst case) of the tabular (basic) implementation of VA is $O(N^2 T)$. The factor $N^2$ is the product of the number of states ($N$) times the maximum in-degree ($N$). However, the maximum in-degree of a lexicon-HMM derived from real languages is in practice independent from $N$, and much smaller than $N$.

The relevant factor is $p = \frac{1}{N}\sum |\text{pred}(j)|$. For example, an actual 130 K words French lexicon [6] gives:

| | $N$ | $p$ |
|---|---|---|
| **trie** | 297701 | 1 |
| **DAWG** | 17908 | 5.22 |

**Table 1**. Lexicons at http://webia.lip6.fr/~lifchitz/FLCVA.

So the average time complexity is $O(NT)$, as for the space one, as summarized below:

| | time (worst) | time (average) | space | best path |
|---|---|---|---|---|
| **tabular** | $O(N^2 T)$ | $O(NT)$ | $O(NT)$ | backtracking |
| **token passing** | $O(N^2 T)$ | $O(NT)$ | $O(N)$ | path history |

**Table 2**. Time and space complexities of the different implementations of VA.

## 3. AN OPTIMAL PATH CODING SYSTEM

The order in which the *full paths* (from the root to the sink) of an automaton are completed in a *Depth First Search* (DFS) [12] provides a canonical indexing of the full paths of the automaton [11]. We call this index the *Perfect Path History* (PPH) as it is a *Minimal Perfect Hashing* [2] and is perfectly suited to the management of path history.

This PPH index is naturally extended to *partial paths* (paths from the root to an internal node), and constitutes an optimal coding scheme for the paths followed by the tokens.

Each partial path $p$ is canonically extended to the full path $\overline{p}$ that is an extension of $p$ and has the smallest PPH index. In other words, $\overline{p}$ extends $p$ by always choosing the first successor to go to the sink. Therefore, we can extend the PPH() function defined originally on the full paths to the partial paths by defining $\mathrm{PPH}(p)$ as $\mathrm{PPH}(\overline{p})$.

In the rest of this section, we show how to compute two variations of the PPH. The first one (forward PPH) considers the successor links whereas the second one (backward PPH) considers the predecessor links.

### 3.1. The forward PPH

Consider a DAWG of a lexicon of $W$ words. For a node $x$ of the automaton, let $\mathrm{suff}(x)$ denotes the number of paths (suffixes) from this node to the sink. In particular $\mathrm{suff}(root) = W$ and $\mathrm{suff}(sink) = 1$. Let $\mathrm{succ}(x,i)$ denote the $i^{th}$ successor. We have the recursive definition:

$$\mathrm{suff}(x) = \begin{cases} 1 & \text{if } x \text{ is the } sink \\ \sum_i \mathrm{suff}(\mathrm{succ}(x,i)) & \text{otherwise} \end{cases}$$

The value of $\mathrm{PPH}(p)$ is by construction in the range $[0, W-1]$ and $\mathrm{PPH}(root) = 0$ as the first full path completed in a DFS has index 0. Let $\mathrm{p}(x)$ be a partial path from the root to a node $x$. We define the *PPH increment* $\Delta\mathrm{PPH}(x, \mathrm{succ}(x,i))$ from $x$ to its $i^{th}$ successor as

$$\Delta\mathrm{PPH}(x, \mathrm{succ}(x,i)) = \sum_{j<i} \mathrm{suff}(\mathrm{succ}(x,j))$$

By construction, we have the following relation on the PPH

$$\mathrm{PPH}(\mathrm{p}(x) + \mathrm{succ}(x,i)) = \mathrm{PPH}(\mathrm{p}(x)) + \Delta\mathrm{PPH}(x, \mathrm{succ}(x,i))$$

The above formula is used to update the token PPH when it is passed from node $x$ to node $\mathrm{succ}(x,i)$.

A useful property of the PPH is that if $p = (x_0, x_1, \ldots, x_k)$ is a full path, then

$$\mathrm{PPH}(p) = \sum_{i=0}^{k-1} \Delta\mathrm{PPH}(x_i, x_{i+1})$$

Given a PPH value $v$ in the range $[0, W-1]$, the corresponding path can be reconstructed as follows,

**Pseudo-code 5**. Path reconstruction from a PPH value

```
// PRE: v is a PPH value in the range [0, W-1]
Set x as the root.
   while x is not the sink
     Determine the largest i such that
      ΔPPH(x, succ(x,i)) ≤ v
     Set v = v − ΔPPH(x, succ(x,i))
     Set x = succ(x,i)
   endwhile
// The nodes visited by x constitute the path
// coded by v.
```

The PPH increment $\Delta\mathrm{PPH}(x, \mathrm{succ}(x,i))$ can either be computed dynamically (minimizing memory space requirement), or can be cached in each arc $x \to \mathrm{succ}(x,i)$ (maximizing speed) as in **Figure 2**. Thanks to its recursive definition, $\mathrm{suff}(x)$ can be computed with a recursive DFS. The complexity of this traversal is linear in the number of nodes. This computation needs to be done only once for a given DAG, during an initialization phase.

The management of the tokens according to SVDE requires only the knowledge of the predecessors of the nodes. However, the forward PPH requires pointers to the successors. In order to only rely on the predecessor information, we consider, in the next sub-section, the forward PPH of the mirror DAWG (where the arcs have been reversed).

### 3.2. The backward PPH

With the backward PPH, we consider paths from the sink to the root. The PPH increment becomes

$$\overline{\Delta\mathrm{PPH}}(x, \mathrm{pred}(x,i)) = \sum_{j<i} \mathrm{pref}(\mathrm{pred}(x,j))$$

where $\mathrm{pref}(x)$ denotes the number of paths (prefixes) from the root to node $x$. Although the backward PPH code paths from the sink to the root, and not from the root to the sink like the forward PPH, the backward PPH is well suited for the coding of full paths from the root to the sink. With

the backward PPH, the sum $\sum_{i=0}^{k-1} \overline{\Delta \mathrm{PPH}}(x_i, x_{i+1})$, where $x_0$ is the sink and $x_k$ is the root, is accumulated starting with $\overline{\Delta \mathrm{PPH}}(x_{k-1}, x_k)$. The backward PPH allows the reconstruction of the mirror path followed by the token during the execution of Viterbi algorithm.

## 4. DISCUSSION AND CONCLUSION

The PPH exhibits interesting properties that makes it a perfectly suited for coding partial paths. In particular, the PPH requires only local information. The size of the PPH values in bits is optimal as it takes its value in the range $[0, W-1]$. Given the PPH value of a token at a given node, it is easy to trace the path followed by the token from the root. The computational complexity is linear in the number of nodes of this path.

In this paper, we have introduced an optimal path coding system for lexically constrained HMM recognition systems using token passing techniques. A lexicon-HMM can benefit from several optimization techniques:

- Compact DAWG in place of traditional tries with simultaneous gain in memory space and running time (typically a ratio 15-20 for a 100 K words lexicon) because of the large reduction of the overall number of HMM states to consider.
- Non deterministic node-automata in place of classical arc-automata for better compacity.
- Reverse topological sort of nodes that halves memory requirement.
- Enhanced $n$-best algorithm.
- Optimal token tagging scheme for path history management in DAWG (forward / backward PPH).

To assess the benefit of using a DAWG over a trie, we ran a test program on the lexicon of 130 K words mentioned in **Table 1**. The test program was designed to evaluate the potential speed-up by simply passing tokens in graphs. We observed an 18 fold reduction in the running time when a DAWG was used instead of a trie.

Future work includes implementing all the techniques described in this paper for lexicon-HMM decoding.

## 5. REFERENCES


[1] R. Bellman, "Dynamic Programming", *Princeton University Press*, 1957.

[2] Z.J. Czech, G. Havas and B.S. Majewski, "Fundamental Study: Perfect Hashing", *Theoretical Computer Science*, Vol. **182**, No 1-2, pp. 1-143, 15 August 1997.

[3] D.G. Forney Jr, "The Viterbi Algorithm", *Proceedings of the IEEE*, Vol. **61**, No 3, pp. 268-278, March 1973.

[4] E. Fredkin, "Trie Memory", *Communications of the ACM*, Vol. **3**, No 9, pp. 490-499, September 1960.

[5] R. Lacouture, R. De Mori, "Lexical Tree Compression", *2nd European Conference on Speech, Communication and Technology (Eurospeech'91)*, Genoa (Italy), pp. 581-584, 24-26 September 1991.

[6] A. Lifchitz and F. Maire, "A Fast Lexically Constrained Viterbi Algorithm for On-Line Handwriting Recognition", *7th International Workshop on Frontiers in Handwriting Recognition (IWFHR7)*, Amsterdam (The Netherlands), pp. 313-322, 11-13 September 2000.

[7] F. Maire, F. Wathne and A. Lifchitz, "Reduction of Non-Deterministic Automata for Hidden Markov Model Based Pattern Recognition Applications", In *"Advances in Artificial Intelligence",* T.D. Gedeon, L.C.C. Fung (eds.), LNCS, Vol. **2903**, pp. 466-476; *Proceedings of 16th Australian Joint Conference on Artificial Intelligence (AI'03),* Perth (Western Australia), 3-5 December 2003.

[8] M. Mohri and M. Riley, "An Efficient Algorithm for the *N*-Best-Strings Problem", *Proceedings of the 7th International Conference on Spoken Language Processing (ICSLP'02)*, Denver (Colorado, USA), 16-20 September 2002.

[9] H. Ney, "Dynamic programming as a technique for pattern recognition", *Proceedings of the IEEE 6th International Conference on Pattern Recognition (ICPR'82)*, Munich (Germany), pp. 1119-1125, October 1982.

[10] L.R. Rabiner and B.-H. Juang, "Fundamentals of Speech Recognition", Ed. Prentice Halls, pp. 321-389, 1993.

[11] D. Revuz, "Dictionnaires et Lexiques, Méthodes et Algorithmes", *Thèse de Doctorat Paris VII (Paris, France), Spécialité: Informatique Fondamentale*, 22 février 1991.

[12] R. Sedgewick, *"Algorithms in C, Part 5. Graph Algorithms"*, *3rd Edition*, 482 p., Addison-Wesley, August 2001.

[13] K.N. Sgarbas, N.D. Fakotakis and G.K. Kokkinakis, "Two Algorithms for Incremental Construction of Directed Acyclic Word Graphs", *International Journal on Artificial Intelligence Tools, World Scientific*, Vol. **4**, No 3, pp. 369-381, 1995.

[14] A.J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm", *IEEE Transactions on Information Theory*, Vol. **13**, pp. 260-269, April 1967.

[15] S.J. Young, N.H. Russell and J.H.S. Thornton, "Token Passing: a Simple Conceptual Model for Connected Speech Recognition Systems", *Cambridge University Engineering Department (England), Tech. Report No TR.38*, 31 July 1989.