Tel-Aviv University
The Raymond and Beverly Sackler
Faculty of Exact Sciences

Efficient Algorithms for Constructing and Employing Variable-Length Markov Models of Language

This thesis is submitted in partial fulfillment
of the requirements for Master degree (M. Sc.)
at Tel-Aviv University
Department of Computer Science

by
Shai Litvak

This thesis has been carried out under the supervision of
Dr. Ron Shamir.

January 1996

Abstract

Statistical models of human language are used as the base for different computer manipulations of textual data: recognition and correction of errors, reconstruction of characters sequences which were coded by an ambiguous code and data compression.

High order Markov models were successfully implemented for English. In some applications the model size is limited. We are then interested in the questions of efficiently constructing an optimal model with a given size, and using it. In this work we give efficient solutions for these two problems.

1. In order to reduce the model size we eliminate some of the less informative parameters which describe dependencies of high orders, and we get a model which has "variable order". We present an algorithm which gets a training sequence and builds a variable order Markov model with a desired size. The model is optimal under the criterion of maximum likelihood with respect to the training sequence among all models with the same size. The algorithm's complexity is quadratic in the desired size. Using this algorithm we have built some models with different sizes for English and demonstrated that they are significantly more informative than standard Markov models with the same sizes. We also present some faster heuristic algorithms for constructing the model and experimental results for using them.
2. We present a dynamic programming algorithm which uses the variable size Markov model to choose a sequence with maximal probability among a set of given sequences. The time and size complexity of the algorithm is linear in the model size and the sequence length. We implemented this algorithm in two applications: the first is used to reduce the error rate in the handwriting recognition system developed by ART company, and the second one decodes a text which was ambiguously encoded using the keys on a standard US phone. We present experimental results for both applications which demonstrate their efficiency as a practical solutions in a commercial quality.

Contents

List of Figures

Acknowledgments

My first gratitude is to my parents which raised me to love learning and to persist on the way for creation.

I warmly thank Dr. Ron Shamir for guiding me on this work. The meeting we had were both intriguing and practical, and introduced me to the practice of the scientific activity. I thank him for his time, for his personal attitude, and for the empathy and encouragement he gave me.

I want to thank ART ltd. and Gabi Ilan for the chance I had to combine the academic and the industrial worlds, and for using the company's equipment and data in the field of handwriting recognition.

I am grateful to Kobi Goldberger for the fruitful conversations, and his important remarks.
I want to thank Dr. E. Boros for contributing a key idea in the "pseudo binary" algorithm, to Dr. S. Skiena, Dr. D. Hochbaum and to Dr. Naftali Tishby for helpful manuscripts.

Special thanks to Moran, for being a partner to my life, to my activities and thoughts. And for her love.

Chapter 1
Introduction

1.1 Language-Like Sequences Modeling: Motivation and Applications

A language-like sequence is a sequence of symbols coming from some natural source and seems to obey some regularity, which is non trivial for capturing. This definition is vague, but the nature of any human language, which is the archetype for the class, is well known to each of us. Other examples for phenomena which exhibits such behavior may be: DNA sequences, protein sequences, daily stock market closing rates or the notes sequence of a musical piece.
Modeling a language-like sequence is the attempt to recognize and formalize patterns and regularities in the phenomena.
The formalization of model for such sequences is meaningful in two aspects:

- It enlarges our knowledge and understanding (and those are, of course, very loaded words) of the phenomena.
- It can be used in a computer program for a variety of interesting applications.

The first aspect can lead to a large variety of works in philosophy, psychology, or brain research. In this work we are interested in the computer application aspect.
The interest in computer application divert the work into two major subjects.
The first one is the model. Some questions rise up immediately: What are the elements of the model and what are their relations ? What is the source data which we base the model upon ? How do we build the model ? How 'good' is the model we get and how can we evaluate it ? Chapters 2 and 3 of this work are trying to answer those questions for the specific models family we suggest, while chapters 4 presents experimental results on test models constructed for English.

The second subject is the use of the model. The range of application is large and it is growing with relation to technology front as well as computer science development. However, we can try to generalize the types of model use in the applications range:

- Using the model for evaluation of phenomena instances. In many recognition tasks the model can be used to choose the best among many interpretations of a given instance. Such tasks can be the recognition of printed or handwritten characters, of sounds or human voice etc. Other application of similar type is automatic error correction of data arriving through a noisy channel. The ambiguity resolution of overloaded symbols transmitted through a low capacity channel can be also achieved using the same technique. An example of such application is the decoding of text which was encoded by a standard ten keys of a phone keypad.
- Using the model as the core of computer representation of phenomena instances. If the model is efficient it can be used for compressing instances and thus lowering their storage space and transmission time.
- Using the model for computer production of new phenomena instances. An example for such application can be the automatic generation of 'melody' based on a model of music.

This thesis focus on the first type of applications. Chapter 5 describes a general technique for evaluation of sequence instance using our model. Chapter 6 presents experimental results of two applications which use the general evaluation technique.

1.2 Statistical Approaches to Modeling Language-Like Sequences

Although human language is probably not generated by a well defined statistical source it does exhibit some declining autocorrelation behavior with respect to time. This means that near past is dominant for future prediction relative to the distant past. The declining function is even more evident in local letter-to-letter autocorrelation. This

suggests that the use of some statistical model for character oriented language modeling will yield good results.
In the following discussion we will present three models which can be used for language modeling and show their benefits and disadvantages:

- Fixed memory length Markov model.
- Hidden Markov model.
- Variable memory length Markov model.

We note that those classical statistical models have been used, not only for language modeling [Nadas 84] but for modeling other 'language-like' sources with the above described feature. This includes, for example, biological sequences such as protein and DNA [Krogh 93].
Other statistical techniques for language modeling are:

- Long distances histories and triggers. See, for example [Huang 93].
- Part of speech tagging. See, for example: [Church 88], [Brill 92] and [Kupiec 92].
- Stochastic context-free grammars. See, for example: [Lari 91], [Schabes 93] and [Jelinek 91].

These models which use a variety of information sources or specific features of human language are out of the scope of this work and are more relevant for word oriented language modeling such that are needed for voice recognition applications.

1.3 The Problems Studied in This Thesis

Two related problems are studied in this work.
The first problem concerns selecting the model used for language-like sequence modeling, and the techniques for constructing such model. We are focusing on a specific family of variable memory length Markov models. Our interest is in finding an optimal model of a desired size with respect to a given training sequence. We use a maximum likelihood approach for defining the optimality criteria of the model. The model is formulated using a weighted tree structure

called 'Probabilistic History Tree' (PHT). We present and discuss the 'Maximum Likelihood PHT' problem in chapter 3.
The second problem concerns using a given variable memory length Markov model for the evaluation of new sequences which are generated by the same source described by the model. More specifically, we are interested in choosing the most likely interpretation for an ambiguous message with respect to our model. The 'Ambiguous Text Resolution' (ATR) problem is discussed chapter 5. We seek a general algorithm for this problem, one that can be used for all the applications which face the same general needs.
A major drive and test for this thesis are the practical applications which we developed based on our theoretical results. The need to meet the technological limitations of small, low cost, portable computers which should run those applications is real-life target for the theoretical work. The wish to find the best possible model of limited size and memory demands reflects in the Maximum Likelihood PHT problem. The limited CPU resources reflects in the Ambiguous Text Resolution algorithm.

1.4 Extant Works and Our Main Results

The discussion in this section is divided to two parts: the Maximum Likelihood PHT problem and the Ambiguous Text Resolution problem. For each problem we present the extant work done and our results.
The class of hypotheses for the variable memory length Markov models which we used is a family of trees we call PHT-s (for details see section 2.5). Each node in the tree represents a state of the Markov chain. The tree is unbalanced and the distance of each node from the root is equal to the memory length of the state. Similar trees have been presented for universal compression purposes by Weinberger, Lempel and Ziv in [Weinberger 82], Rissanen [Rissanen 83], and developed by Weinberger, Rissanen and Feder [Weinberger 95].

The same class of models was studied by Ron, Singer and Tishby in [Ron 95] and used for correcting English text. Their aim was learning a variable memory length Markov model which they represent by a "probabilistic finite state automaton" (PFSA). They gave a PAC-like learning algorithm for doing this, with the following features: If both a bound L on the memory length of the target PFSA, and a bound n on the number of states of the target PFSA has are known, then for every given $\varepsilon>0$ and $0<\delta<1$, their learning algorithm outputs an ε-good PHT (using the Kullback-Leibler divergence to measure the error with respect to the target PFSA), with confidence parameter $1-\delta$, in time polynomial in L, n, $|\sigma|$ (the alphabet size), $1/\varepsilon$ and $1/\delta$. The algorithm is fed with a long enough training set which was produced by the target PFSA, and with the parameters L, n, ε and δ. Ron et al also show how to construct a PHT with $N = L \cdot n$ nodes which is equivalent to a given n-state PFSA.

A similar PAC model learnability problem was also studied in [Höffgen 93].

Our approach to the problem is a bit different from the PAC learning approach. We measure the likelihood of our hypotheses with respect to the training set. Our aim is to find a maximum likelihood PHT with a desired number of nodes. In chapter 3 we show that the maximum likelihood PHT problem can be reformulated as a rooted maximal weighted subtree problem. The original tree describes the full L order Markov model or some subtree which is most likely to contain the desired maximum likelihood PHT. We analyze the general maximum weighted subtree problem and give two algorithms for solving it. One algorithm is greedy and does not give an optimal solution, yet, test results shows that its solution is near optimal. The other algorithm finds the optimal solution in not more than $c\cdot|V|\cdot N^2$ were c is constant, V is the nodes set of the original tree and N is the number of desired nodes in the target subtree. After the completion of our work on the algorithm, we received a preprint by Goldschmidt and Hochbaum [Goldschmidt 95] which studies the k-edge subgraph problem. They give an algorithm for the weighted subtree problem which is rather similar to our algorithm.

We present a third algorithm for the weighted subgraph problem which takes advantage of the inner order which is most likely to be found among the tree nodes, due to the special weights derived from the maximum likelihood PHT problem. In the original tree the weight of the node is almost always higher than the weight of each of its children. This order is used to speed up the search for the optimal solution.

We used our three algorithms to build a set of PHT-s of different sizes for English. We show these PHT-s and compare their features in chapter 4.

Learning a variable length Markov model is only the first step needed for successful use of this model in a computerized recognition application. Given a sequence of ambiguous signs produced by a known source, where each sign has several possible interpretations, we can use our knowledge of the source to pick the 'best' interpretation for each sign. The algorithm presented by Viterbi [Viterbi 67] is very useful for this task when the source is modeled using a fixed order Markov model. This simple dynamic programming algorithm was implemented for a variety of applications such as speech recognition [Rabiner 86], optical characters recognition [Hull 82] and phone keypad-based communication [Skiena 94].

Faster algorithms for doing the same task were presented by Fano [Fano 63] and Jelinek [Jelinek 69]. They are based on scanning only a portion of all the possible combinations of interpretations with high probability of finding the best combination. They do not ensure, however, that the best combination will be found, which is the case in Viterbi algorithm.

There are some works which extend the Viterbi algorithm for use with variable length dependencies models. Tao [Tao 92] describes a generalization of hidden Markov model and of Viterbi algorithm which uses a building blocks of short, variable length sequences of characters. His generalized Viterbi algorithm runs in time $O(N^2)$ for interpreting one sign when N is the number of states in the model. Tao does not give a learning algorithm for constructing the model.

Ron, Singer and Tishby [Ron 95] version of the Viterbi algorithm is based on the automaton description of the variable length Markov

model. This algorithm needs $O(|\sigma|\cdot N)$ operations per sign when N is the number of states in the model and σ is the alphabet used.
Our algorithm is basically an expansion of the Viterbi algorithm which uses a PHT model instead of the fixed order Markov model. The algorithm has an upper bound of $c\cdot|\sigma|\cdot N$ operations per sign, however, the actual work per node is far beneath this bound. The algorithm is carefully designed for on-line work in a limited platform of small low cost portable computer. Extra care was taken for speed. We present a technique which uses a subset of reasonable hypotheses per sign (which is naturally given in many real life applications) to reduce processing time. Special care was taken also for minimizing memory consumption, and for the on-line behavior of the algorithm, that is, the algorithm is buffering signs on-line and resolves them as-soon-as-possible while more signs are coming. This algorithm is used in the commercial handwriting recognition system of ART company which is running on a limited platform. Another application which was based on this algorithm is an American phone keypad based communication system. On the transmitting side the keypad is used to encode English messages with an ambiguous code of ten digits. On the receiving side the ambiguous message is being resolved with high accuracy using our algorithm.

Chapter 2
Statistical Models For Language-Like Sequences

In this chapter we will present three types of statistical models for language-like sequences and point out their benefits and disadvantages:

- Fixed memory length Markov model.
- Hidden Markov model.
- Variable memory length Markov model.

Two preliminary issues should be discussed before facing the specific problem of learning some language-like model: The training set and the model quality.

2.1 Information Source: The Training Set

When we approach the problem of modeling a language we first need some stream of characters coming from the relevant source. The stream is used to set model parameters, and should be a good representative of the source. This means it should be long enough to contain the needed statistics and as general as possible in it contents. The selection of such a stream for English modeling is not a simple task since any stream has its particularities:

- It may originate from either written text or spoken text.
- It is usually discussing one or some specific subjects which have their own dominant vocabulary and abbreviations. Specific names that are repeatedly appearing might insert noise to our model. The Bible, or Alice in Wonderland, for example, can make some names too popular in the model, if used as training sets.
- It can represent different layers of English. Clinton speeches, for example, might be more useful than Shakespeare's tragedies for today's applications.

- It tends to exhibit some specific format (especially if it is written and edited somehow). An unformatted text, delivered through phone keypad, for example, is very different from a written formatted text of a paper dictionary.
- It is influenced by the specific speaker or writer jargon.

The application for which the model is built can take advantage of the particularities if it is aimed toward specific user or situation. In this case, we can consider using some non-typical English training set, one which was produced under the conditions our model is designed to work in.

This is not the case, however, when the application is more general-purpose. In this case some combined training set, built up from some sources containing a variety of authors, subjects, formats etc. should be composed.

Another parameter that should be considered when deciding on a training set is the special psychological situation that is typical to the man-machine interface of the application which uses the model. In case of on-line phone keypad based communication, for example, the users might develop special habits like short words or basic vocabulary use, or even generating short non-grammatical "sentences" with no conjunctions. Collecting data from the real situation in which the application should be used could be helpful for building a good model. It is not always easy, however, to collect such real data since phone keypad based communication or on-line character recognition are not yet world-wide popular.

2.2 Measuring Model Quality

Measuring model quality is needed whenever one of several models (or parameters sets for a model) should be chosen for a specific application. If the model is designed to work in a specific application then the ultimate measure for the model quality is its overall contribution to the performance of this application.

In practice, this approach is usually quite problematic:

- It is not always easy to measure application performance itself. To measure the error rate of the application, a large data collected from a variety of real users is needed. It is usually not handy or easy to collect.
- Another problem with measuring the application error rate is that it is not necessarily equivalent to human evaluation of the application. For example, some phenomena among the errors, although not very common, might not be tolerated by the users.
- Error rates depend on many parameters of the application. Those parameters are sometimes combined non-linearly with the model, and separating the model error from the error in the application parameters is not always possible. For example, in the case of handwriting recognition, the language model parameters and the shape model parameters are combined and influence the application error rates together.
- Tuning the model for a specific application makes the model non-general purpose.

Focusing on the model itself and on the way it is being used within the application to assign probabilities for character sequences is more practical, and easier to analyze. Optimization of the model will usually lead to improvement in overall application performance.

In order to build a model we need some training set. Suppose $X=c_1...c_R$ is a stream of characters generated by the source that we wish to learn. Each c_i is a character in σ, the source alphabet. If X is long enough and represents this source's general behavior, then it can be used as a good training set. A representative training set can also be used for estimating model quality in the following manner:

Let M be some candidate stochastic model with memory L (see section 2.3). M defines a probability measure $P_M(d|d_1..d_L)$ for each $d,d_1..d_L \in \sigma$ for the event "d is next to appear given that the last L characters in X were $d_1..d_L$". We are interested in evaluating the quality of M with respect to X. The *likelihood L of M with respect to X* is defined:

$$L_X(M) \equiv P_M(X) = \prod_{i=L+1}^{R} P_M(c_i | c_{i-L}..c_{i-1})$$

(For simplicity we will ignore, for now, the first L characters. This will be further discussed in section 3.1.)

When using the *maximal likelihood approach* one should pick the model M for which $L_X(M)$ is maximal. The intuition behind this approach is that since X is a typical string from the source, picking a model which gives high probability to the appearance of X is more reasonable than picking a model which predicts the appearance of X with small probability. A mathematical justification for this approach is by a simple Bayes law argument:

$$P(M|X) = \frac{P(X|M)P(M)}{P(X)}$$

P(X) is fixed for all models. The a-priori probability P(M) is equal for all models since we have no a-priori preference to any model. This means that a model M maximizing $P(X|M)$ will also maximize $P(M|X)$. The maximum likelihood approach has been widely used for voice recognition [Bhal 83] and other recognition and compression tasks.

Let $P_X(d_1...d_L)$ be the probability measure of the sequence $d_1..d_L$ in X, that is, the proportion of $d_1..d_L$ among all L-long strings (L successive characters) in X.

We can also use the data-to-model *cross-entropy H(X;M)* measure, known also as *logprob* [Jelinek 89], instead of the likelihood measure:

$$H(X;M) \equiv -\sum_{\substack{\text{all } d_1...d_{L+1} \\ d_i \in \sigma}} P_X(d_1...d_L) \cdot \log(P_M(d_{L+1}|d_1...d_L)) = ... = -\frac{1}{R}\log(L_X(M))$$

It is clear that minimizing $H(X;M)$ is equivalent to maximizing $L_X(M)$ while cross-entropy is sometimes more convenient to analyze and measure.

Another equivalent measure called *perplexity* [Jelinek 77] is used as well in the literature to measure model with respect to the text:

$$S_M(X) = 2^{H(X;M)}$$

2.3 Markov Models

The basic stochastic model which we will describe here is the Markov model. We also introduce the probabilistic automaton model and its relation to the Markov model.

Stochastic processes

A *stochastic process* is an indexed sequence of random variables. The set of possible values which the random variable may assume is called the *state space* of the process. If the index set I of the stochastic process $\{X_i\}_{i\in I}$ is countable then the process is called *discrete time stochastic process.* X_t is then called the *state* of the process *in time t.*

A discrete stochastic process is *stationary* if:

$\Pr(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = \Pr(X_{1+k} = x_1, X_{2+k} = x_2, \ldots, X_{n+k} = x_n)$

for every k=1,2... and all $x_1, x_2, \ldots, x_n$ in the state space of the process.

For the stationary process we use the notation:

$$p(x_1, x_2, \ldots, x_n) = \Pr(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n).$$

First order Markov processes

A discrete stochastic process (X_i) is a *first order Markov process* or a *Markov chain* if for all $x_1, x_2, \ldots, x_n$:

$$p(x_n|x_1, x_2, \ldots, x_{n-2}, x_{n-1}) = p(x_n|x_{n-1})$$

In this case we can write the chain form:

$$p(x_1, x_2, \ldots, x_n) = p(x_1)p(x_2|x_1)p(x_3|x_2), \ldots, p(x_n|x_{n-1})$$

Stationary first order Markov processes

When for the Markov process $\Pr(X_{n+1} = x_i|X_n = x_j)$ is independent of n then the process is called *time invariant.* Note that a stationary process is time invariant but the converse is not necessarily true. In this case we can define:

$$p_{i,j} = \Pr(X_{n+1} = x_i|X_n = x_j).$$

Since we can write $p(x_1, x_2, \ldots, x_n)$ in a chain form than any time invariant first order Markov process is characterized by the set $\{p_{i,j},\ i, j \in I\}$ and the distribution of the initial state X_1.

A probability distribution $\{p_j, j=1,2,\ldots\}$ is called a *stationary distribution* for the transition probabilities $\{p_{i,j}\}$ if:

$$p_j = \sum_{i \in I} p_i p_{i,j}$$

Theorem 2.1:
If the distribution of the initial state X_1 of a first order Markov process is a stationary distribution then the process is stationary.

Another known theorem in stochastic processes theory states:

Theorem 2.2:
Suppose the first order Markov process satisfies the following conditions:

- There exist a stationary distribution.
- The process is *irreducible*, that is, the probability that starting from state x_i the process will eventually get to state x_j is non-zero, for every states x_i and x_j.
- The process is *aperiodic*, that is, $p_{i,i} > 0$ for every state x_i.

then the stationary distribution is **unique**.

This means that a stationary Markov process is characterized merely by the set $\{p_{i,j},\ i, j \in I\}$ which also defines X_1's distribution.

The proof for theorems 2.1 and 2.2 can be found in literature (see, for example [Ross 70] chapter 4).

K-th order Markov processes
We can define Markov processes of higher order. If the process satisfies the condition:

$$p(x_n|x_1,\ldots,x_{n-2},x_{n-1}) = p(x_n|x_{n-k},\ldots,x_{n-2},x_{n-1})$$

then it said to be *k-th order Markov process*. The order is also called the *memory* of process.
A generalization of the theorems describing the stationarity conditions (2.1 and 2.2) can be also given for the k-th order Markov process.

We can view a Markov model from another point as well. It can be described as a random walk on a probabilistic automaton.

Probabilistic Finite Automaton

A *PFA, probabilistic Finite Automaton* is characterized by the 5-tuple (Q,σ,δ,P,P_0) defined as follows:

- Q is a finite set of states.
- σ is a finite alphabet.
- $\delta: Q\times\sigma\to Q$ is a transition functions.
- The transition probability functions $P: Q\times\sigma\to[0,1]$. P(q,c) is the probability of stepping from state q to state δ(q,c). P must satisfy the following conditions: for every $q\in Q$: $\sum_{c\in\sigma} P(q,c)=1$.
- The probability distribution $P_0: Q\to[0,1]$ of the first state. $P_0(q)$ is the probability that the initial state will be q. P_0 must satisfy the following conditions: $\sum_{c\in\sigma} P_0(c)=1$.

A PFA can be used to simulate a time invariant Markov process of order k and space state X by defining:

- $Q\equiv X^k$
- $\sigma\equiv X$
- for every $x\in\sigma$, $x_1x_2\ldots x_k\in Q$, let $\delta(x_1x_2\ldots x_k,x)=x_2\ldots x_kx$
- for every $x\in\sigma$, $x_1x_2\ldots x_k\in Q$, let :
  $P(x_1x_2\ldots x_k,x)\equiv \Pr\{X_n=x|X_{n-k}=x_1,\ldots,X_{n-1}=x_k\}$
- let $P_0(x_1x_2\ldots x_k)$ be $\Pr(X_1=x_1,X_2=x_2,\ldots,X_k=x_k)$, the distribution of the initial k letters, for every $x_1x_2\ldots x_k\in Q$.

Notice that if $P_0(x_1x_2\ldots x_k)=p_{x_1x_2\ldots x_k}$ is a stationary distribution of the Markov process, then the process is stationary.

Example:

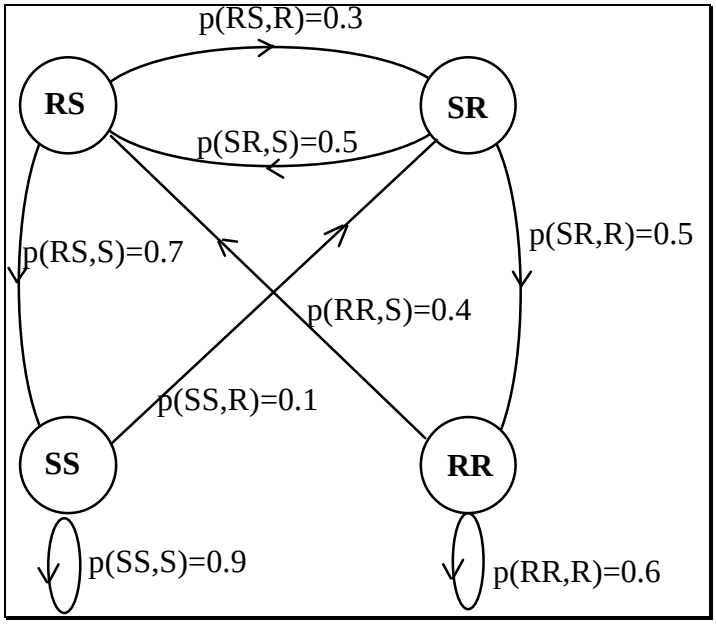


Figure 2.1: A simple second order Markov model of Israel's weather is presented as a PFA. We define $P_0(x) \equiv P(SS,x)$. *The alphabet is R (Rain) and S (Sun). There are four states (SS, RR, SR and RS). The model can be used for weather forecast based on the last two days weather. Rain is rare in general, especially after two days of sun. The states RS and SR occur only in the winter, and indeed the chance for another day of rain increases. Rain is most probable after two days of rain. In contrast with Hidden Markov Model which will be defined later, the states have a natural meaning and they define all the possible histories of length 2.*

Several descriptions of English as a Markov process were introduced by Shannon in his famous paper [Shannon 48]. Although English is probably not an ergodic stationary process, it seems to be well approximated by such descriptions. Shannon defined several models for English which differed by the state space and the model order. He presented a sequence of letters models with growing order, as well as single words, word pairs and word triplets models. Here are two examples from Shannon's paper describing outputs of Markov models for English with different state space:

- Fourth order letters model:

THE GENERATED JOB PROVIDUAL BETTER TRAND THE DISPLAYED CODE, ABOVERY UPONDULTS WELL THE CODERST IN THESTICAL IT DO HOCK BOTH MERG....

- Second order word model:
  THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER METHOD OR THE LETTERS...

The examples were generated by simulating the Markov process, that is, by making a random choice (with appropriate probability) of letters (or words) preceded by the previous state in the generated sequence. One major difference between the models is their size, that is, the number of transition probabilities in the models. The single letter model is of size 27 (counting all the letters and the space character). The letter quadruplets model is of size $27^4 = 531441$, while the word pairs model size rise up to millions of combinations. As the model size grows, its resemblance to the English language is more evident. Still, high order Markov models involve the manipulation of a large set of parameters. This is not feasible on a small computer running an on-line application due to limitations of both time and memory. It also seems that many of the parameters are redundant and most long history probabilities are actually not informative.

2.4 Hidden Markov Models

We now describe the Hidden Markov Model (HMM). This model is more general (and thus stronger) than the Markov model.

Definitions:

Let (X_i) be a Markov process. Each state X_i is associated with a new random variable Y_i. The sequence of random variables (Y_i) which are associated with (X_i) respectively is called a *hidden Markov process.*

The transitions between the states of this stochastic process are done using the underlying Markov model (X_i), but what we observe at each state is only the associated random variable Y_i.
The assumption that the underlying Markov model is stationary is very common in most practical implementation of this model.
The inherent difference between this model and the simple Markov model is that in the hidden Markov process the underlying Markov process is indeed "hidden". That is, the Markov states of the process are not observed directly.
Many real world processes perform such a behavior and can be efficiently described by an HMM. The HMM has been widely used in many applications such as speech recognition [Rabiner 89] and handwriting recognition [Chen 92].
If the Markov chain (X_i) is stationary so is the corresponding hidden Markov (Y_i) since the transition between the states of the hidden process is still defined by the underlying Markov process.

Example:

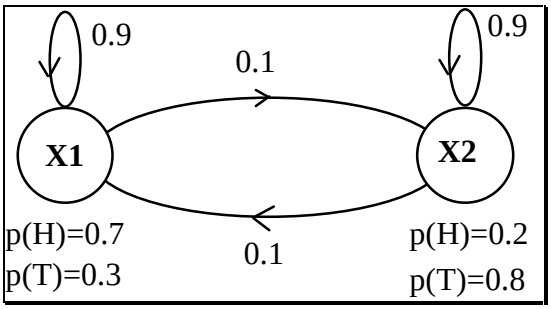


Figure 2.2: A Hidden Markov Model with two states. The process tends to stay at a state when it gets there. X1 is biased towards H and X2 is biased towards T.

This Hidden Markov Process can describe the following scenario:
A man is hidden behind a curtain and announces the results of his actions. The results are a sequence of T-s and H-s. The actions that leads to this results are not visible to us.
Now, what the man actually does is this: At each "round" he first flips a biased coin C with p(T)=0.9 and p(H)=0.1. He uses the result to move between two tables X1 and X2. If he gets T then he stays at the

table he had been in last round. If he gets H he moves to the other table. At each table there is another unbalanced coin. Coin C1 at table X1 is biased toward H, while C2 at table X2 is biased towards T. After the man decides, based on C result, what is the table for this round he flips the coin at that table and tells us the result.
Notice that the underlying Markov model that makes him move is not directly visible to us. Yet, the typical sequence produced by the described source will contain long consecutive subsequences each of which is dominated by one results (T or H). This hints that there are two states in the process and that one is biased towards T and the other towards H.

The question of learning a HMM from a training set is not simple since the correlation between the observed training set and the model states is not direct. The number of states and the structure of the models are unknown and can't be guessed naturally from the given training set. There is no known analytic way to solve the maximization problem of learning the maximum likelihood (see section 2.2) HMM with respect to the training set. Few answers which involves a converging scheme were proposed [Rabiner 86]. However, it had been proved [Abe 92] that there is no polynomial time algorithm (in the alphabet size) for learning HMM, unless RP=NP. Moreover, even when the alphabet size is fixed there is no known algorithm for that task which is polynomial in the size of the target HMM.
When modeling a language for character based applications it seems that the choice of the last n characters as a meaningful state of the model is reasonable. A good evidence for this choice is the fact that Markov models of high order for the language simulate it rather well, at least when focusing on the local scope of characters in the sentence (see examples in 2.3). If such a choice is done the sequence of last n observed characters explicitly determines the state in which the model is currently, and the use of HMM is unnecessary.

2.5 Variable Memory Length Markov Models

For simplicity we shall use a binary alphabet in the following discussion. However, the results are general and remain valid for any alphabet size.
In a typical situation which we try to deal with, we are given a stream $\overline{X}=c_1...c_{\overline{R}}$ of characters, $c_i \in (0,1)$, all generated from a single source.
In general we are looking for a compact model that could have generated the given stream with high probability. Later we will use this model to estimate the probability of other strings which are coming from a source similar to the one that generated $\overline{X}$. We will limit the discussion to a reasonable family of models defined below.

Definitions:
Let $\sigma=(0,1)$ be an alphabet and let $\sigma_{\leq L}$ be the set of all strings with maximal length L over σ.
For a rooted tree T=(V,E) let *root(T)* be the tree's root. For every node $v \in V$ let *children(v)* be the set of v's children and *des(v)* the set of v's descendants.
A *history tree, HT* T=(V,E,S) over $\sigma_{\leq L}$ is a binary tree (V,E) with an associated 1-1 function $S:V \rightarrow \sigma_{\leq L}$. $S(root(T))=\varepsilon$ and for every other $v \in V$: $S(v)=c \bullet S(parent(v))$, for some $c \in \sigma$, $c \neq \varepsilon$, and "$\bullet$" stands for concatenation.
With the notation $suf(c_1...c_i)=c_2...c_i$, $suf(c)=\varepsilon$ and $suf(\varepsilon)=\varepsilon$, this can be written also as: if $v' \in$ children(v) then $suf(S(v'))=S(v)$.

Example:

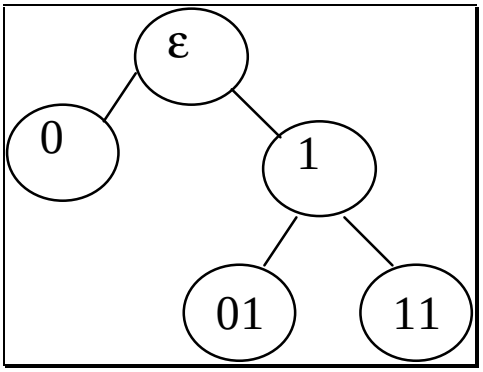


Figure 2.3: A simple binary HT. In order to find the longest suffix of the string $s=c_1 \ldots c_k$ start at the root. If $c_k=0$ then head left, otherwise turn right. Proceed with c_{k-1}, c_{k-2}... until c_1 or a leaf is reached.

A *PHT, probabilistic history tree* $T=(V,E,S,\Re)$ is a history tree with a set $\Re=\{P_v\}_{v\in V}$ of probability functions, $\forall v \in V$, $P_v:\sigma\rightarrow[0,1]$.

A PHT can be used as source to generate a character stream in the following manner:

- Suppose $c_1 \ldots c_{i-1}$ is the stream of characters already generated by the source.
- Let $v_i \in V$ be the node such that $S(v_i) = c_{i-j} \ldots c_{i-1}$ and j is maximal (for i=1 let v_1 be root(T)).
- Choose the value of c_i at random according to probability distribution P_{v_i}.

We define here a way to use a PHT for assigning probability to any string $s=c_1 \ldots c_r$:

$$P(s)=\prod_{i=1}^{r} P_{v_i}(c_i)$$

where $v_i \in V$ is the node such that $S(v_i) = c_{i-j} \ldots c_{i-1}$ and j is maximal (for i=1: v_1 is root(T)). If $P_{v_i}(c_i)$ is chosen to be the probability $P(c_i|S(v_i))$ in the training set $\overline{X}$ then this value is the best evaluation we maintain in our model for the precise value $P(c_i|c_1 \ldots c_{i-1})$. Notice

that if we knew these precise probabilities $P(c_i|c_1\ldots c_{i-1})$ for each c_i, i=1..r, then we could calculate P(s) by:

$$P(s) = P(c_1)\cdot P(c_2|c_1)\cdot\ldots\cdot P(c_r|c_1\ldots c_{r-1})$$

In chapter 3 we show how to construct a PHT and a corresponding set $\{P_v\}_{v\in V}$ which will give us best evaluation for P(s) based on our training set $\overline{X}$.

Example:

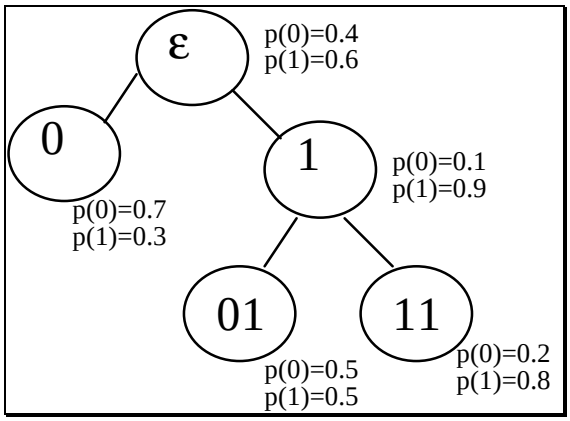


Figure 2.4: A simple PHT. There is a probability function P_v associated with each node v, describing the probability of the next character provided that the suffix of the already seen characters series is S(v).

Probability evaluation for the string 1011 using the above PHT is done like this:

Define $Q(c_1\ldots c_i)$ to be the node corresponding to the longest suffix of $c_1\ldots c_{i-1}$ which appears in T. In our case this makes:

Q(1)=ε,	P(1|ε)=0.6
Q(10)=1	P(0|1)=0.1
Q(101)=0	P(1|0)=0.3
Q(1011)=01	P(1|01)=0.5

and the desired probability is:

$$P(1011)=0.6\cdot 0.1\cdot 0.3\cdot 0.5 = 0.009.$$

Let $depth(T)\equiv \max_{v\in V}(|S(v)|)$. Notice that after k=depth(T) characters the inner nodes of the tree are no longer in use and the source can be

viewed as a Markov source of order k, in which some (or many) of the k-tuple probabilities are not explicitly part of the model.
The probability function of the Markovian state which is associated with the string $c_1 \ldots c_i$ is P_v when v is the node such that $S(v)=c_{i-j} \ldots c_i$, $j \leq i$ and j is maximal.
Since the complete description of a general, order L Markov model is of size $O(|\sigma|^{L+1})$ it is quite impractical to maintain it for growing L-s. Still, when modeling a language-like source there are cases where the 'deep' past is meaningful. A Markov model using a PHT description may contain data associated with variable length histories and can keep a lot of meaningful data with a relatively small number of parameters. In a PHT description for the Markov model each set of "uninteresting" long histories is represented by only one node whose string is the common suffix of all those histories. More interesting histories get their own nodes. The problem is, of course, to decide which are the interesting histories.

We will use the notation *probabilistic history tree machine* M_T for a model based on a PHT T used for evaluating probabilities in the method described above.

To measure the model quality we will use the likelihood of M_T with respect to our training set. Let L be some upper bound to depth(T). Since $\overline{X}=c_1 \ldots c_{\overline{R}}$ is typically long we can measure the likelihood with respect to $X=c_{L+1} \ldots c_{\overline{R}}$ instead and get almost the same model. Giving up those few characters in the beginning of the set will make our calculations simpler.
The likelihood of such model M_T with respect to X is:

$$L_X(M_T) = \prod_{i=L+1}^{\overline{R}} P_{v_i}(c_i)$$

where each v_i is defined by T.

A main problem which we will discuss in the next chapter is:

Problem 2.1 - Maximum Likelihood PHT of Bounded Size:
Given are a sequence of characters $\overline{X}=c_1...c_{\overline{R}}$, $c_i \in \sigma$, and positive integers L and N. Find a PHT T with depth(T)≤ L and |V|≤N such that $L_X(M_T)$ is maximal.

Chapter 3
Algorithms for Constructing Maximum Likelihood Probabilistic History Tree

We ended chapter 2 exhibiting problem 2.1 - PHT Likelihood Maximization. In this chapter we will describe some attempts to give exact or near-optimal solutions to this problem.
First we will generalize the problem into a weighted tree maximization problem. Then we will analyze the general tree problem. At the end of the chapter we will discuss the special case of the original problem.

3.1 Reformulation as Weighted Subtree Maximization Problem

Here is a reminder of problem 2.1 from section 2.5.

Problem 2.1 - Maximum Likelihood PHT of Bounded Size:
Given are a sequence of characters $\overline{X}=c_1...c_{\overline{R}}$, $c_i \in \sigma$, and positive integers L and N. Find a PHT T with depth(T)≤ L and |V|≤N such that $L_X(M_T)$ is maximal.

Recall that $X=c_{L+1}...c_{\overline{R}}$ is almost identical to $\overline{X}$, with only the first L characters missing. Neglecting this relatively small number of characters ($L<<\overline{R}$) simplifies the following calculations without changing statistics significantly. We will use the notation $R = \overline{R} - L$.

Definitions:
Let T=(V,H,S) be a history tree (HT), and let $\overline{X}$ be the training set.
Defined below are three functions, all named C with different parameters. They can be easily distinguished by the parameters used or by context:

1. $C(s\cdot c):\sigma_{\le L}\times\sigma \mid\rightarrow N$. $C(s\cdot c)$ is the number of c_i-s in X such that c_i=c, and c_i is preceded by s ($s\in\sigma_{\le L}$). Note that c_i is in X but s may be in $\overline{X}$, so every c_i is preceded by at least L characters.
2. C(v,c): $V\times\sigma \mid\rightarrow N$. It is defined using 1: $C(v,c)\equiv C(S(v)\cdot c)$.
3. C(v): $V \mid\rightarrow N$. It is defined using 2: $C(v)\equiv\sum_{c\in\sigma}C(v,c)$.

We described in section 2.5 the way M_T is used for evaluating the probability of X, $P(X|M_T)$. In view of this process the functions C(v,c) and C(v) have the following meaning:

- For all leaves of T, C(v,c) is the number of times in the process that $P_v(c)$ was used to evaluate c's apostriori probability.
- For all leaves of T, C(v) is the overall number of times $P_v()$ was used in the process to evaluate any character's apostriori probability.

We can use C(v,c) and C(v) to define a natural set of probability functions, one for every node $v\in V$:

$\overline{P}_{T,v}:\sigma \mid\rightarrow [0,1]$. $\overline{P}_{T,v}(c)\equiv\frac{C(v,c)}{C(v)}$

Another probability function is:

$P_T: V\times\sigma \mid\rightarrow [0,1]$. $P_T(v,c)\equiv\frac{C(v,c)}{R}$. $P_T(v,c)$ is the probability of the event: "node v was used to evaluate the probability of character c", which is counted by C(v,c).

The following theorem is well known for Markov models of any fixed memory length. For the completeness of this chapter we prove it for the special case of the PHT based model.

Theorem 3.1:
Let (V,E,S) be a history tree, and let $\overline{X}$ be the training set.
A PHT T=(V,E,S,$\Re$) that maximizes $L_X(M_T)$ must satisfy:
$P_v(c)=\overline{P}_{T,v}(c)$ for all leaves of T.

Proof:

We give the proof for the case of binary alphabet $\sigma = \{0,1\}$. It can be generalized for alphabet of any size by following the same proof line.

The inner nodes of the tree are not used at all when calculating $P(X|M_T)$. This is since we are evaluating characters only from the L+1 position in the set, and each one is preceded by at least L others. Probabilities are evaluated by the nodes of T with the longest strings, so, the leaves are always preferred. Hence, we have to optimize only the probability functions of the leaves. Let $v_1 \ldots v_b$ be the leaves of T.

In the proof we will use the notation p_i for $P_{v_i}(0)$.

In order to maximize $L_X(M_T)$ we write it as a function *f* of *b* parameters:

$$L_X(M_T) = P(X|M_T) = f(p_1, p_2, \ldots, p_b) = \prod_{\substack{i=1..b \\ C(v_i,0)>0}} p_i^{C(v_i,0)} \prod_{\substack{i=1..b \\ C(v_i,1)>0}} (1-p_i)^{C(v_i,1)}$$

It is equivalent and easier to maximize log(f(...)):

$$g(p_1, \ldots, p_b) = \log(f(p_1, \ldots, p_b)) = \sum_{\substack{i=1..b \\ C(v_i,0)>0}} C(v_i,0)\log(p_i) + \sum_{\substack{i=1..b \\ C(v_i,1)>0}} C(v_i,1)\log(1-p_i)$$

Since *g* is a sum of terms, each containing a single parameter, we can find separately the maximum of each parameter to get the global maxima. Consider the terms involving the parameter p_i:

$$C(v_i,0)\log(p_i) + C(v_i,1)\log(1-p_i)$$

There are four cases:

1. If $C(v_i,0)=0$ and $C(v_i,1)>0$ then clearly $p_i=0$ maximizes the sum since p_i appears only in the right term.
2. If $C(v_i,0)>0$ and $C(v_i,1)=0$ then clearly $p_i=1$ maximizes the sum since p_i appears only in the left term.
3. If $C(v_i,0)>0$ and $C(v_i,1))>0$ then we can assume $0<p_i<1$. The assumption is justified since $p_i=0$ implies $L_X(M_T)=0$ while $p_i=1$ means that $P_{v_i}(1)=0$ and again $L_X(M_T)=0$.

$$\frac{dg}{dp_i} = \frac{C(v_i,0)}{p_i} - \frac{C(v_i,1)}{1-p_i} = 0$$

hence:

$$P_{v_i}(0) = p_i = \frac{C(v_i,0)}{C(v_i,0)+C(v_i,1)} = \overline{P}_{T,v_i}(0)$$

it is also easy to check that:

$$\frac{d^2g}{dp_i^2}\Big|_{\frac{C(v_i,0)}{C(v_i,0)+C(v_i,1)}} < 0$$

and so a maximum solution was found.

4. If both $c(v_i,0)=0$ and $c(v_i,1)=0$ the value of p_i does not effect the sum and can be chosen arbitrarily.

$P_{v_i}(1)$ is defined by: $P_{v_i}(1)=1-p_i$.♦

We have found that the probability functions of the inner nodes have no effect of the likelihood maximization process, and we can choose them arbitrarily. This is since we choose to ignore the beginning of the sequence which has a minor effect on the global model. However, when using the model for real life applications, we often face the beginning of a new short sequence which we cannot ignore. If we want to evaluate probabilities of characters in the beginning of such sequence we have to use the inner nodes probabilities. We can choose those probabilities such that the model will keep its stationary behavior at the beginning of short strings as well, by defining $P_v(c)=\overline{P}_{T,v}(c)$ for the inner nodes as we did for the leaves.

In the following calculation we will use the data-to-model *cross-entropy* measure instead of likelihood measure.

$$H(X;M_T) = -\frac{1}{R}\log(L_X(M_T))$$

Hence, minimizing $H(X;M_T)$ is equivalent to maximizing $L_X(M_T)$. This can be expressed also as:

$$= -\frac{1}{R}\log(\prod_{i=L+1}^{\overline{R}} P_{v_i}(c_i)) = -\frac{1}{R}\sum_{i=L+1}^{\overline{R}} \log(P_{v_i}(c_i))$$

and using the definition of $P_T(v,c)$ we can get:

$$H(X;M_T) = -\sum_{\substack{v\in leaves(T)\\ c\in\sigma}} P_T(v,c)\log(P_v(c))$$

Taking any HT T=(V,E,S) we will analyze the change in the minimal cross-entropy when "splitting" one of T's leaves $\bar{v}$ into its $|\sigma|$ children, producing a new HT T′.
We have two PHT models: the original $M_T=(V,E,S,\overline{\Re})$ and the expanded $M_{T'}=(V',E',S',\overline{\Re}')$. $\overline{\Re}$ and $\overline{\Re}'$ are the maximum likelihood solutions of the two history trees, respectively.

Example:

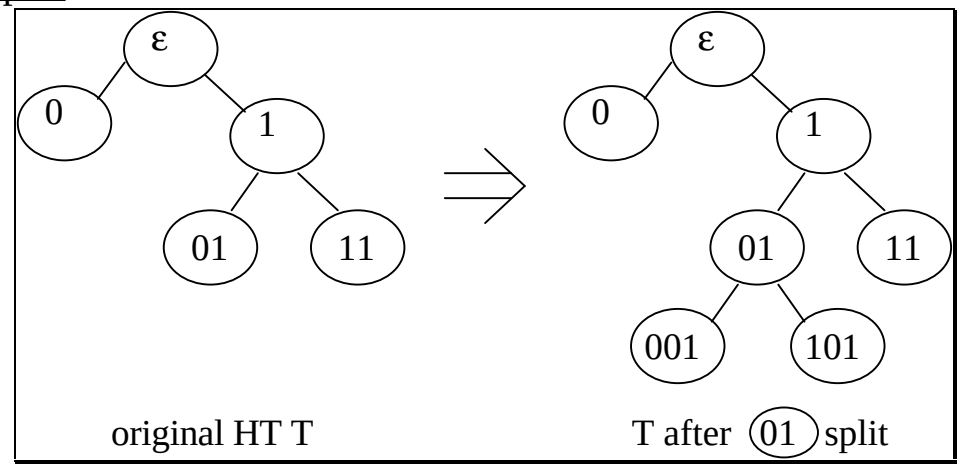


Figure 3.1: One split of a HT node. The leaf "01" is split and the new PHT has two new leaves: "001" and "101".

The change of cross entropy between the models is:

$$H(X;M_T) - H(X;M_{T'}) =$$

$$\sum_{\substack{v\in leaves(T')\\ c\in\sigma}} P_{T'}(v,c)\log(P_{T',v}(c)) - \sum_{\substack{v\in leaves(T)\\ c\in\sigma}} P_T(v,c)\log(P_{T,v}(c)) =$$

Based on the result of theorem 3.1 the maximum likelihood probability functions sets $\overline{\Re}$ and $\overline{\Re}'$ are be chosen such that $P_{T,v}(c)=\overline{P}_{T,v}(c)$ for all nodes T. This means that:

$$H(X;M_T) - H(X;M_{T'}) =$$
$$= \sum_{\substack{v \in leaves(T') \\ c \in \sigma}} P_{T'}(v,c) \log(\overline{P}_{T',v}(c)) - \sum_{\substack{v \in leaves(T) \\ c \in \sigma}} P_T(v,c) \log(\overline{P}_{T,v}(c)) =$$

$$= \sum_{c \in \sigma} [\sum_{j \in \sigma} P_{T'}(v_j,c) \log(\overline{P}_{T',v_j}(c)) - P_T(\overline{v},c) \log(\overline{P}_{T,\overline{v}}(c))] =$$
$$= \frac{1}{R} \sum_{c \in \sigma} [\sum_{j \in \sigma} C(v_j,c) \log \frac{C(v_j,c)}{C(v_j)} - C(\overline{v},c) \log \frac{C(\overline{v},c)}{C(\overline{v})}]$$

Denoting $e(v,c) = C(v,c) \log \frac{C(v,c)}{C(v)}$ we get:

$$H(X;M_T) - H(X;M_{T'}) = \frac{1}{R} \sum_{c \in \sigma} [\sum_{j \in \sigma} e(v_j,c) - e(\overline{v},c)]$$

This indicates that the cross-entropy change made by the split does not depend on the whole original HT and is only a **local function** of the specific split.

Now we can define:

$$\Delta_s \equiv \frac{1}{R} \sum_{c \in \sigma} [\sum_{j \in \sigma} e(v_j,c) - e(\overline{v},c)]$$

(assuming X is our original given stream), to be the cross-entropy reduction achieved when splitting properly a leaf $\overline{v}$ with an associated string s of any history tree T to its children. Known results from information theory (see, for example, [Cover 91] chapter 2) imply that since after the split we have more information in our model then $\Delta_s \geq 0$.

Problem 3.1 - Weighted Binary HT Subtree Maximization:
Let T=(V,E,S) be some HT with depth(T)≤L. Let $w_v=\Delta_{s(v)}$ be the weight of the node v for every v∈V. Find a subtree $\overline{T}$ of T, rooted at root(T), with $|V(\overline{T})|=N_W$ such that the total weight of nodes in $\overline{T}$ is maximum.

Theorem 3.2:
Problem 2.1 can be polynomially reduced to problem 3.1.

Proof:
Given an instance of problem 2.1 with integers L and N, construct the following instance for problem 3.1:

- let L be the same value from the instance of problem 2.1.
- let $N_W = \frac{N-1}{|\sigma|}$.
- let T is the full HT of order L.
- The weights $w_v=\Delta_{s(v)}$ calculated for the set of probability functions suggested in theorem 3.1.

A solution to problem 3.1 is translated to solution for problem 2.1 by taking $\overline{T}$ and adding all direct children of $\overline{T}$ leaves. $\overline{\Re}$ should be the corresponding functions set given by theorem 3.1. The maximum weight of $\overline{T}$ ensures maximum cross entropy reduction relative to the empty rooted subtree {ε}, and thus gives the minimum cross entropy PHT model (with respect to X).
Since the weights are all positive ($\Delta_s \geq 0$) we are ensured to get maximum solution using exactly N nodes.♦

3.2 Algorithms for Weighted Subtree Maximization Problem

We first discuss a more general subtree maximization problem. The special problem in which the weights correspond to PHT problem will be discussed in section 3.3 .
We assume that all trees and subtrees are rooted at r (the root node, or the ε node in the PHT model). The *weight of a tree* is the sum of weights of its nodes.

Problem 3.2 - Weighted Subtree Maximization:
Given are a rooted tree T=(V, E), a weight function w(v), w:V→R and a positive integer N≤|V|. Find a rooted subtree $\overline{T}_N$ of T with N nodes and maximum weight.

We will first describe a greedy algorithm for solving this problem and then analyze its solutions.

Greedy subtree maximization algorithm

$V_1=\{\varepsilon\}$
i=1
while i < N do:
    let $C(V,V_i)$ be the set of all V's nodes that are direct children of V_i's leaves.
    $\overline{v} = \arg\max_{v\in C(V,V_i)}(w(v))$
    $V_{i+1} = V_i \cup \{\overline{v}\}$
    i=i+1
output V_N

The output of the algorithm is the tree $T_N=(V_N, E_N)$. T_N is the subtree induced by V_N.

Theorem 3.3:
If the degree of T is bounded by D then the greedy algorithm can be

implemented to run in time complexity of O(N(logN+D)).

Proof:
Since the 'while' statement repeats N time, it suffices to show that each iteration can be performed in O(logN+D). This can be achieved by keeping an updated array A[v] of nodes defined as follows.
After step i, let potential(i,v) be the set of direct children of $v \in V_i$ which may potentially be added in the step i+1. Now, for every v such that potential(i,v)$\neq\varnothing$ a single node x satisfying:

$$x=A[v]=\arg\max_{v' \in Potential(i,v)} (w(v'))$$

is kept in A. A is kept ordered by decreasing weights of A[v]. At the i-th step there are i nodes in V_i, and for each one there is at most one array entry A[v] holding the 'heaviest' child of v not yet in V_i, if such a child exist. Hence, the array size is $|A| \leq i$.
Clearly $\arg\max_{v \in C(V,V_i)}(w(v))$ is the first node in A, and so, having A ordered facilitates finding $\bar{v}$ in O(1) time.
To keep A ordered we need to update it after adding a node $\bar{v}$ to V_i as a child of v: First, deleting the old A[v] takes O(1).
Then, we find the new A[v] and A[$\bar{v}$] which can be done in O(D).
Finally, we add A[v] and A[$\bar{v}$] to A in their correct positions which takes O(logN) since $|A| \leq i \leq N$. ♦

Definition:
A weight function on the nodes of the tree is called *monotone* if along any path from the root to a leaf, the weights are non-increasing.

Theorem 3.4:
Let T be a tree, and let f be a **monotone weight function** on its nodes. Then the greedy algorithm gives an **optimal solution** to the subtree maximization problem.

proof:
Let $T_N=(V_N, E_N)$ be a solution generated by the greedy algorithm, and let $\bar{T}_N=(\bar{V}_N, \bar{E}_N)$ be some optimal solution with N nodes. We will

show, by repeating a process of node replacement, how to generate a new maximal solution which has more nodes in common with T_N, without decreasing its weight. The process ends when we find that our solution is optimal as well. In each stage we will replace one node in $\overline{T}_N$ by a node from T_N.

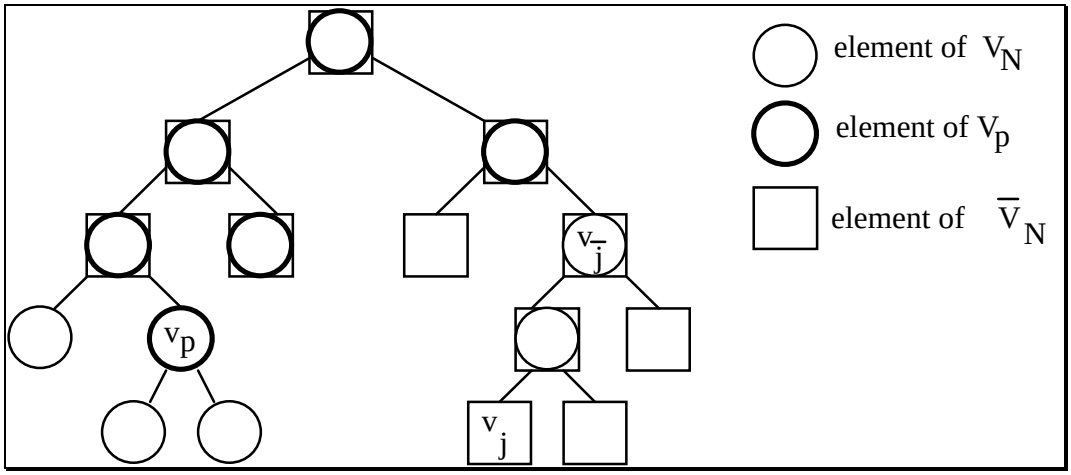


Figure 3.2: Scheme of the replacement

Let v_p be the first node in V_N which does not belong to $\overline{V}_N$, with respect to the order of choice of vertices in the greedy algorithm.
Denote by V_{p-1} the set of nodes in the greedy subtree after p-1 iterations.
Let v_j be any leaf of $\overline{T}_N$ which does not belong to V_N. Let $v_{\bar{j}}$ be the last node in the path from v_j to the root which does not belong to V_{p-1}. At step p, vertex v_p was chosen by the greedy algorithm and $v_{\bar{j}}$ was not, so $w(v_{\bar{j}}) \le w(v_p)$. The monotonicity implies $w(v_j) \le w(v_{\bar{j}})$, hence $w(v_j) \le w(v_p)$. By setting $V_N \leftarrow V_N \setminus \{v_j\} \cup \{v_P\}$ we get a new tree with equal or greater weight which differs from T_N by one less vertex.
We repeat the replacement process until $V_N = \overline{V}_N$, and we conclude that the greedy algorithm had found an optimal solution. ♦

The fast greedy algorithm works well for monotone trees. But we need an algorithm that works for any tree. We will suggest such one. In order to make the discussion simple we will first deal with simplified version that works for binary trees, and then expand it to a general algorithm for any tree.

Problem 3.3 - Weighted Binary Subtree Maximization:

Find a rooted subtree T^N_{max} with N nodes and a maximum total weight of a given complete weighted binary tree T.

Notation:

Define l(v) and r(v) to be the left and right children of any v in T.
We shall assume the convention that l(v) and r(v) correspond to extending v's strings by 0 and 1, respectively. Define g(v) to be the level of v, that is, for all leaves g(v)=1, for all their parents g(v)=2, etc. Let g(root)=G.

The following algorithm computes the function t(v,i). This function is defined for each node v in T and for each i which is at most the size of the complete subtree rooted at v. t(v,i) is the maximal weight of any i nodes subtree rooted at v.

Dynamic programming algorithm for Weighted Binary Subtree Maximization problem

Calculate the following function t(v,i):

$\forall v \in T: \qquad t(v,0) = 0$

$\forall v \in T, \text{ if } v \text{ is a leaf}: \quad t(v,1) = w(v)$

$\forall v \in T, \text{if } v \text{ is not a leaf}, \ \forall i = 1..2^{g(v)} - 1:$

$$t(v,i) = w(v) + \max_{\substack{j,k \in \{0..2^{g(v)-1}-1\} \\ j+k+1=i}} [t(l(v),j) + t(r(v),k)]$$

The weight of the T_{max}^{N} is t(root,N). By maintaining for each pair (v,i) the values j and k which achieve the maximum value t(v,i), the optimal subtree can subsequently be constructed.

Theorem 3.5:
The algorithm runs in time complexity of $O(|V|^2)$.

Proof:
Once all the values t(v,i) and the maximizing j,k are known, the construction of the tree takes O(1) per node (of the subtree), hence, a total of O(N) time. The major effort is calculating the values t(v,i). The analysis is simple since T is binary:

- Let us compute the total work required to calculate t(v,i) for $i=0..2^{g(v)}-1$ and a single node v with g(v)=g: Every pair j,k contributes to exactly one i, namely i=j+k+1. Hence, the total number of sums is the multiplication of the sizes of the left and right subtrees, i.e. $2^{g-1} \cdot 2^{g-1} = 2^{2g-2}$.
- The number of nodes at level g is 2^{G-g}. Hence, the total work in level g is $2^{G-g} \cdot 2^{2g-2} = 2^{G+g-2}$.
- The total work in the tree is therefore
  $\sum_{g=1}^{G} 2^{G+g-2} = 2^{G-1} \cdot (2^G - 1) = 2^{2G-1} - 2^{G-1}$.

Since $|V|=2^G-1$ the total time complexity of finding the maximum weighted subtree is $O(|V|^2)$. ♦

A glance over the algorithm shows that there might be some unnecessary work done. Take the case N=2, with only 2 possible subtrees, or N=|V|-1 with at most |V| possible subtrees. These examples show how N imposes restrictions on the requested subtree structure and hence on the number of calculations needed. We will show that those restrictions can be used in the algorithm and reduce the work for extreme values of N. However, the worst case complexity does not change.

The following version of the algorithm has two additional features over the previous version:

- It is designed for any tree T.
- It uses N to control unnecessary calculations of t(v,i).

Notation:

size(v) is the number of vertices in the subtree rooted at v.
children(v) is the set of children of v.
D = $\max_{v \in V}(|\text{children}(v)|)$.

The algorithm below computes t(v,i), the same function as in the binary tree problem. least(v) and most(v) are two new functions which we use to limit the calculations of t(v,i) only to the necessary arguments. least(v) is a lower bound on the size of the subtree rooted at v in the solution to the maximization problem with size N. most(v) is the maximum possible size of this subtree.

Dynamic programming algorithm for General Weighted Subtree Maximization problem

Calculate the following functions:

least(v),most(v):

least(root)=most(root)=N

$\forall v \in T$, if v is not the root:

$\text{least}(v)=\max\{0,\ \text{size}(v) + \text{least}(\text{father}(v)) - \text{size}(\text{father}(v))\}$

$\text{most}(v)=\min\{\text{size}(v),\ \text{most}(\text{father}(v)) - 1\}$

t(v,i) :

$\forall v \in T$: $t(v,0) = 0$

$\forall v \in T$, if v is a leaf: $t(v,1) = w(v)$

$$\forall v \in T, \text{if } v \text{ is not a leaf}, \ \forall i = \max(least(v),1)...most(v):$$

$$t(v,i) = \max_{\substack{least(\bar{v}) \le j_{\bar{v}} \le most(\bar{v}) \\ \sum_{\bar{v} \in children(v)} j_{\bar{v}} = i-1}} [\sum_{\bar{v} \in children(v)} t(\bar{v}, j_{\bar{v}})] + w(v)$$

The maximum weighted subtree is constructed in the same way as in the binary tree algorithm. That is, by maintaining the $j_{\bar{v}}$-s found for the maximum t(v,i).

Theorem 3.6:

If there is a node $u \in T$ whose distance from the root is k such that $\forall v \in children(u)$: least(v)=0 and most(v)=N-k-1, then the time complexity W for calculating t(root,N) is bounded:

$$c_1 D\left(\frac{N+D-k-2}{D-1}\right)^{D-1} \le W \le c_2 |V| N^D$$

Proof:

lower bound:

t(u,N-k) is the maximum sum in a set of sums. Each sum is computed in O(D) time. The number of sums in the set is the number of possibilities of dividing N-k-1 equal elements among D cells since each of the D children can have from 0 to N-k-1 nodes. The number of combinations is:

$$\left[\begin{matrix} N-k-1+D-1 \\ N-k-1 \end{matrix}\right]$$

Hence, the total time for computing t(u,N-k) (when N-k is large with respect to D) is at least: $c_1 D\left(\frac{N+D-k-2}{D-1}\right)^{D-1}$.

upper bound:

The maximum work for calculating all needed t(v′,i) for one node v′ is proportional to the total number of subtrees combinations tested.

Since $\min_{\bar{v}\in T}(\text{least}(\bar{v})) = 0$ and $\max_{\bar{v}\in T}(\text{most}(\bar{v})) \le N$ then the maximum total number of subtrees combinations per node is bounded by N^D and the total work for all nodes is bounded by $c_2|V|N^D$.♦

Conclusion (for English HT model):
The conditions in theorem 3.6 hold in the typical case of the optimal PHT model problem. In this case D=27 and the size of the original tree can be 10^3 to 10^5 (depending on the model order) from which we might want to prune 90-99% in order to get a PHT which holds most of the model information and is still feasible for use on a small, relatively slow machine. The useful range of N is usually few hundreds to few thousands of nodes. The root node usually satisfies the condition in theorem 3.6 for these typical sizes. This means that for the English model the maximal likelihood subtree PHT is not computable in reasonable time using the dynamic programming algorithm.

'Pseudo Binary' Algorithm for the General Weighted Subtree Maximization problem

Since in practice N≥1000, the use of the algorithm with D=27 is impractical. However, for D=2 the time complexity is bounded by $c_2|V|N^2$ operations which is much more practical. If we could avoid the exponential effect of D on the time complexity then our algorithm would become feasible for any general tree.
Here is a general scheme of the way to do this:

- Transform the general tree to an equivalent binary tree which is not much larger.
- Apply the dynamic programming algorithm on the binary tree, and transform the solution into a solution for the original problem.

The 'pseudo binary' algorithm is composed of two procedures.
The following recursive procedure *make_bin(v)* is called with the root of the original general tree. It creates a new binary tree composed of the original nodes plus new 'pseudo' nodes. The new tree structure is

presented by the attributes l(v) and r(v) which are set by the procedure for all the inner nodes of the binary tree (which may be original or pseudo nodes). The weights of the original nodes are kept unchanged and the weights of all pseudo nodes are defined to be zero.

```
procedure make_bin(v)
begin
        if v is a leaf then return.

        let r(v)← the rightmost child of v.
        make_bin(r(v)).

        if v has just one child then
        begin
                let l(v) ← null.
                return
        end

        if v has exactly  two children then let l(v)← the left child of v.
        else begin
                create a new 'pseudo' node u and set w(u)← 0.
                delete all subtrees rooted at each child of v but r(v).
                hang all these subtrees on u.
                let l(v) ← u.
        end

        make_bin(l(v))
end
```

Here is a simple example of an original tree and the corresponding binary tree generated by make_bin.

Example:

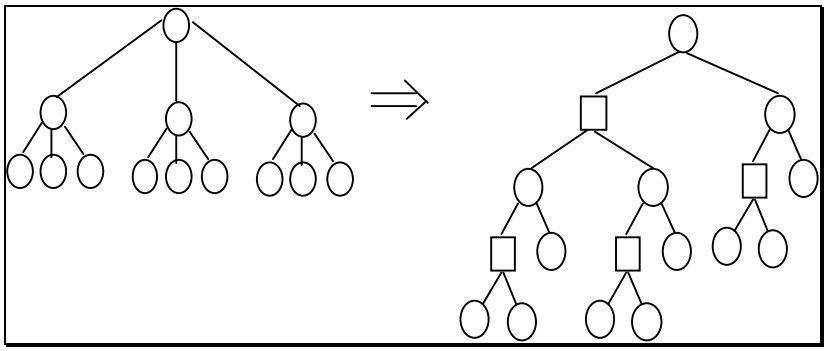


Figure 3.3: The ternary tree on the left is transformed by make_bin to the binary tree on the right. The round nodes are original nodes and the square ones are new 'pseudo' nodes. Notice that the new tree is deeper but not much larger the original tree.

The number of nodes in the new tree is not more than twice the number in the original tree. The argument is simple. At each time make_bin is entered (without immediately exiting in the first line) we can mark one previously unmarked node, r(v), in the original tree. Since only one new pseudo node can be created during one make_bin call it is clear that the overall number of pseudo nodes is not larger than the number of the original nodes.
The procedure takes O(size(T)) steps. This is since the number of recursive calls is not more than size(T) while each call can be implemented in O(1).

The second step in the algorithm is finding solution to the binary tree maximization problem and translating the solution back to the original tree problem. These two actions can be combined in one algorithm which is almost identical to the original dynamic programming algorithm for binary trees presented earlier in this section.
The algorithm computes the function t(v,i), which has a bit different meaning from the original algorithm. t(v,i) is the maximal weight of

any subtree rooted at v and containing exactly i **original** (not pseudo) nodes.
A new function orig(v) is computed. It describes the number of original nodes which are descendants of v (including v itself). orig(v) is only an assisting function in the computation of t(v,i).

Calculate the following functions t(v,i) and orig(v):

$\forall v \in T$: $\quad t(v,0) = 0$

$\forall v \in T$, if v is a leaf: $\quad t(v,1) = w(v)$

$\quad orig(v) = 1$

$\forall v \in T$, if v is not a leaf, and v is an **original** node:

$$orig(v)=orig(l(v))+orig(r(v))+1$$

$\forall i = 1..orig(v)$:

$$t(v,i) = w(v) + \max_{\substack{j \in \{0..orig(l(v))\} \\ k \in \{0..orig(r(v))\} \\ j+k+1=i}} [t(l(v),j) + t(r(v),k)]$$

$\forall v \in T$, if v is not a leaf, and v is a **pseudo** node:

$$orig(v)=orig(l(v))+orig(r(v))$$

$\forall i = 1..orig(v)$:

$$t(v,i) = \max_{\substack{j \in \{0..orig(l(v))\} \\ k \in \{0..orig(r(v))\} \\ j+k=i}} [t(l(v),j) + t(r(v),k)]$$

The weight of the T^N_{max} is t(root,N). By maintaining the values j and k which achieve the maximum value t(v,i) for each pair (v,i), the optimal binary subtree containing N original nodes can subsequently be constructed. All that is left to be done now is choose the same

nodes in the original tree, **ignoring** all the pseudo node in the binary tree which are zero weighted anyway, and do not influence the maximization. Comparing the original and the binary tree clears the fact that all possible combinations of 'original subtrees' have actually been tested while maximizing the binary tree, giving the optimal result for the general tree problem.
Since the binary tree size is not more than twice the size of the original tree we found the desired solution in not more than $c_3|V|N^2$ operations.

3.3 A Hybrid Algorithm for Subtree Maximization Problem

We have seen two types of algorithms for weighted subtree maximization problem. One is the basic dynamic programming algorithm with time complexity of about $O(|V|N^D)$ and its "pseudo binary" version running at $O(|V|N^2)$. The other is a greedy algorithm which finds the solution for monotone trees only, but with time complexity of only O(N(logN+D)). The inner order in monotone trees lets us speed up the solution finding dramatically.
Now, suppose we have a non-monotone tree T with some incomplete but meaningful inner order.

Example:

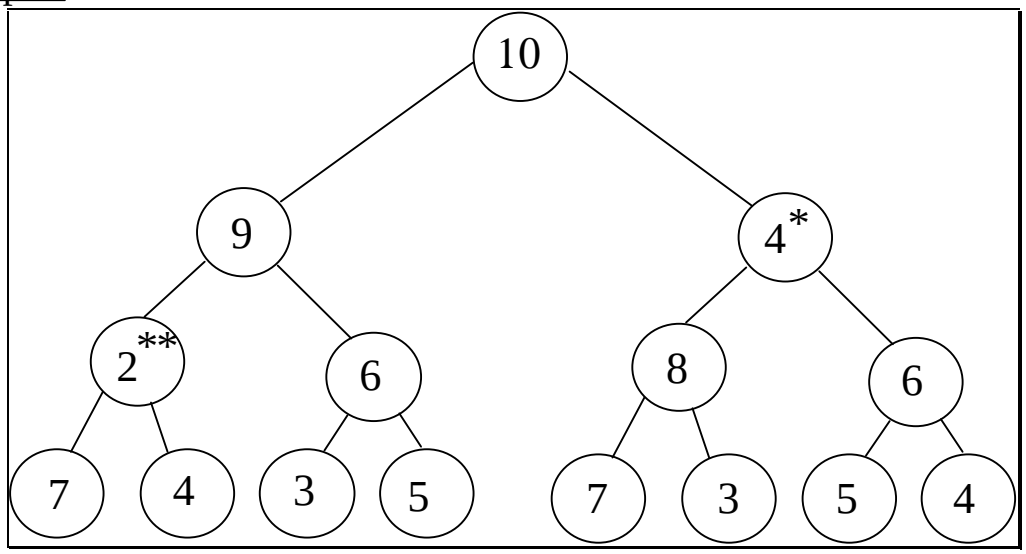


*Figure 3.4: A "nearly monotone" tree. Changing the weights in only two nodes (those marked with * and **) can make this tree monotone.*

Is it possible to exploit the 'almost' monotonicity features of the tree to get a fast solution to the problem ?
The answer is yes. The following hybrid algorithm is a careful mixture of the greedy and dynamic programming algorithms introduced before. It correctly solves the maximization problem for **any** tree. Trees with no inner order at all are treated in dynamic programming approach, while trees with 'islands' of order are treated more like monotone trees with a greedy method.

Hybrid algorithm for Weighted Subtree Maximization Problem

Calculate the following functions starting from the root and progressing recursively to each of the subtrees hanged on the root:

least(v),most(v):

least(root)=most(root)=N

$\forall v \in T$, if v is not the root:

$least(v)=\max\{0,\ size(v)+least(fater(v))-size(father(v))\}$

$most(v)=\min\{size(v),\ most(father(v))-1\}$

Calculate the two parameters function t(v,i) starting from the leaves and stepping up recursively toward the root:

t(v,i):

$\forall v \in T$: $t(v,0) = 0$

$\forall v \in T$, if v is a leaf: $t(v,1) = w(v)$

$\forall v \in T$, if v is not a leaf:

Perform the following preprocessing step, that speeds up t(v,i) maximization in the next step:

- Let $v_1 \ldots v_m$ be the children of v.
- For every k=1..m, j=1..size(v_k): Let $d_{k,j} = t(v_k, j) - t(v_k, j-1)$.
- From each sequence $d_{k,1} \ldots d_{k,size(v_k)}$ take out a minimum number n_k of elements $d_{k,r_{k,1}} \ldots d_{k,r_{k,n_k}}$ such that the sequence of the remaining elements is monotone non-increasing.
  Let $r_{k,0} = 0$ $\forall$k=1..m.
- Let $\Phi = (r_{1,0} \ldots r_{1,n_1}) \times (r_{2,0} \ldots r_{2,n_2}) \times \ldots \times (r_{m,0} \ldots r_{m,n_m})$.
- For each $R = (r_1, r_2, \ldots, r_m) \in \Phi$ perform the following greedy algorithm with the respective arrays $S_R[]$ and $I_R[]$:
  a. Define $next(r_i) = r_{i,j+1}$ if $r_i = r_{i,j}, j < n_i$ and $next(r_i) = size(v_i)$ if $r_i = r_{i,n_i}$.
  b. For every k=1..m let Front[k]=r_k+1
  c. item=$1 + \sum_{k=1..m} r_k$
  d. While there exist k such that Front[k]< next(r_k) do
     1. $i = \arg\max_{1 \le k \le m \text{ and } Front[k] < next(r_k)} (d_{k,Front[k]})$
     2. $S_R[item] = d_{i,Front[i]}$.
     3. $I_R[item] = i$
     4. item=item+1
     5. Front[i]=Front[i]+1

$\forall i = \max(least(v),1)\ldots most(v)$:

$$t(v,i) = \max_{\substack{\text{all } (r_1\ldots r_m)=R\in\Phi \\ \text{such that:} \\ \Sigma r_k \le i-1 \le \Sigma next(r_k)}} \left[\sum_{k=1..m} t(v_k, r_k) + \sum_{j=(1+\Sigma r_k)\ldots(i-1)} S_R[j] \right] + w(v)$$

After each t(v,i) maximization, use the chosen R to calculate the two parameters function best(v_k,i):

$\forall k=1..m,\ \forall i = \max(least(v),1)\ldots most(v)$:

$$best(v_k,i)=r_k+|\{ j \mid I_R(j) = k,\ \ 1+\sum_{r_u\in R} r_u \le j \le i-1\}|$$

The maximum weighted subtree itself is found at the end of process by recursively calculating the following function use() starting from the root:

- use(root)=N
- For every node v which is not a leaf:
  Let $v_1\ldots v_m$ be the children of v, define -
  use(v_k)=best(v_k,use(v)).

The solution subtree contains all nodes v such that use(v)≠0.

Theorem 3.7:
The Hybrid algorithm finds a maximum weighted subtree of T.

Proof:
This algorithm is similar to the general dynamic programming algorithm and the logic of the dynamic programming scheme is identical. The only new part which needs to be proved is the special maximization stage used in the calculations of t(v,i) for one node.
Let $v_1\ldots v_m$ be the children of v. For simplicity we will assume that for all k=1..m all the values $t(v_k,0)\ \ldots\ t(v_k,size(v_k))$ are known (while in practice the unneeded values are not calculated.)

We will prove that:
given

$$t(v_1,0)\ \ldots\ t(v_1,size(v_1))$$
$$\vdots$$
$$t(v_m,0)\ \ldots\ t(v_m,size(v_m))$$

the maximum sum of m items $t(v_1,r_1),...,t(v_m,r_m)$, one from each line, such that $\sum_{k=1..m} r_k = i-1$, is t(v,i)-w(v) as calculated by the algorithm.

Let us assume that the maximum sum $\bar{t}(v,i)$ is a total of the following m items:

$$t(v_1,\bar{j}_1)\ \ldots\ t(v_m,\bar{j}_m) \text{ where } \sum_{k=1..m} \bar{j}_k = i-1.$$

In terms of the preprocessing stage in the algorithm, this is also the sum of:

$$d_{1,1}\ \ldots\ d_{1,\bar{j}_1}$$
$$\vdots$$
$$d_{m,1}\ \ldots\ d_{m,\bar{j}_m}$$

since we defined $d_{k,j} = t(v_k,j) - t(v_k,j-1)$.
We will show that the above algorithm finds a sum t(v,i) such that $\bar{t}(v,i) \le t(v,i)$.
Let $\bar{r}_k = \max\{r_{k,h} | r_{k,h} \le \bar{j}_k,\ 0 \le h \le n_k\}$ for every k=1...m.
The $r_{k,h}$-s and the n_k-s are those defined in the preprocessing stage.
Let $\bar{R} = (\bar{r}_1 \ldots \bar{r}_n)$ be the corresponding vector from the preprocessing stage.
Let $S_{\bar{R}}$ be the associated array. The vector $\bar{R}$ with the array $S_{\bar{R}}$ were checked (along with other vectors) by the algorithm in the process of calculating t(v,i).

Define: $f = 1 + \sum_{k=1..m} \bar{r}_k$

$G = \{S_{\bar{R}}[f] \ldots S_{\bar{R}}[i-1]\}$

$\bar{G} = \{d_{k,j} | 1 \le k \le m,\ \bar{r}_k + 1 \le j \le \bar{j}_k\}$

G is a set of $d_{k,j}$-s tested by our algorithm while maximizing t(v,i). $\bar{G}$ is the respective set from the optimal solution.

Using definition of $S_{\bar{R}}[f] \ldots S_{\bar{R}}[i-1]$ from the algorithm we can define the set of indexes $j_1 \ldots j_m$ by viewing G as follows:

$$G = \{d_{k,j} | 1 \le k \le m,\ \bar{r}_k + 1 \le j \text{ and } j \le \mathbf{j_k}\}$$

Since G and $\bar{G}$ must have the same size it derives that $\sum_{k=1..m} j_k = \sum_{k=1..m} \bar{j}_k$.

Notice also that for each k=1..m the $\bar{r}_k$-s were chosen such that $d_{k,\bar{r}_k+1} \le d_{k,\bar{r}_k+2} \le \ldots \le d_{k,\bar{j}_k}$.

Consider two solutions: Let $\bar{t}(v,i)$ be an optimal solution, and let $t^*(v,i)$ be the value of the term corresponding to vector $\bar{R}$ in computing t(v,i) by the algorithm. That value is considered in the maximization process. Each value is a sum of some $d_{k,j}$-s.

These solutions $\bar{t}(v,i)$ and $t^*(v,i)$ might differ only in the $d_{k,j}$- s indexed $\bar{r}_k + 1 \le j$. We will show, by repeating a process of local replacement of $\bar{G}$'s elements by others of G, that our solution is a maximal one as well.

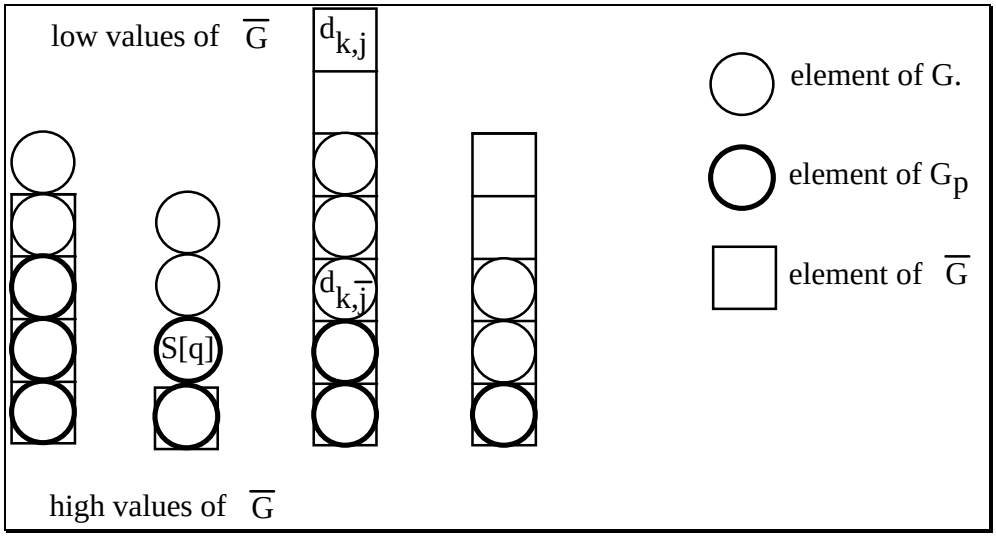


Figure 3.5: Scheme of the replacement

Let S[q] be the first element in the sequence $S_{\overline{R}}[f] \ldots S_{\overline{R}}[i-1]$ which does not belong to $\overline{G}$. If there is no such element then G=$\overline{G}$ and we have nothing to prove.
Otherwise, define $G_p = \{S_{\overline{R}}[f] \ldots S_{\overline{R}}[q]\}$.
Take any $d_{k,j} \in \overline{G} \backslash G$ with maximal index j.
Take $d_{k,\bar{j}} \in \overline{G} \backslash G_p$ (same k) with minimal index $\bar{j}$.
$S[q] \geq d_{k,\bar{j}}$ since step 1 in the preprocessing stage chooses the maximal d possible at each stage.
$d_{k,\bar{j}} \geq d_{k,j}$ by the monotonicity among $\overline{G}$'s elements for every k.
So, $S[q] \geq d_{k,j}$, but then we could replace $d_{k,j}$ with S[q], and get a new optimal solution which has more d-s in common with G.
We repeat the replacement process until G=$\overline{G}$. Hence our algorithm indeed tests and finds an optimal solution when maximizing t(v,i).♦

computational complexity of hybrid algorithm

The evaluation of the computational complexity of the algorithm in the general case is not simple. However, it is possible to analyze it for extreme cases: The monotone tree and the completely unordered tree.
The common part of algorithm which is identical in all cases is the search for the non-monotone elements. For each node $v_k \in V$ we have to find the longest monotone decreasing subsequence for the sequence $d_{k,1} \ldots d_{k,size(v_k)}$. An algorithm for doing this was given by Orlowski and Pachter in 1989 [Orlowski 89]. The computational complexity of the algorithm is $O(m \cdot \log(n))$ where m is the size of the original sequence and n is the size of the subsequence. This means that the total work for all the nodes is limited by:

$$c_1 \cdot \sum_{v \in V} size(v) \log(size(v)) \leq c_1 \cdot \log(|V|) \sum_{v \in V} size(v)$$

Since we can ignore all items indexed more than N in the original sequences by using most(v) this work is limited by:

$$c_1 \cdot \log(N) \sum_{v \in V} size(v)$$

When summing up size(v) for all nodes in V we can use the fact that each node is counted once by each of its ancestors. This gives us an upper bound:

$$\sum_{v \in V} size(v) \leq depth(T)|V|$$

In summary, the total work invested in calculating all the longest subsequences is bounded by:

$$c_1 \cdot depth(T) \cdot |V| \cdot \log(N)$$

We can evaluate the total work of the algorithm in three cases:

1. If for each node v there is just one vector R in the maximization stage then the greedy algorithm works just once per node and the time is $O(size(v) \cdot D)$. Using the above argument this sums up to $O(depth(T) \cdot |V| \cdot D)$ over all nodes of T. The total work for the complete algorithm is then: $O(depth(T) \cdot |V| \cdot [\log(N) + D])$.

   Notice that this is not much larger than the computational complexity of the greedy algorithm from chapter 3.2 which is $O(N \cdot [\log(N) + D])$.
2. If for each node v there is no order found for all the sequences $d_{k,1} \ldots d_{k,size(v_k)}$ then the maximization process always evaluates all the combinations of dividing i-1 among the node's children. Notice that when there is complete disorder $next(r_k) = r_k + 1$. Since the preprocessing stage stops when for every k=1..m Front[k]$\geq next(r_k)$ then the loop is never entered and the computational complexity for each vector R is O(1). We find that in this case the hybrid algorithm works similar to the simple dynamic programming algorithm and its overall computational complexity is equal to that of the dynamic programming algorithm.
3. If we have a bound parameter ϕ for the size of Φ we can evaluate the amount of work in the general case. For each R in Φ the preprocessing stage takes no more than $O(size(v) \cdot D)$. This will make the overall work be no more than:

   $O(depth(T) \cdot |V| \cdot [\log(N) + D \cdot \phi])$.

In the typical case of using the algorithm for creating an English PHT the parameters in the last formula are more specific:

- |V| is usually in the range 10^3 to10^5, and the tree is more or less balanced.
- N is a small fraction of |V| (e.g. 10%)
- D is very small relative to |V| and N. It is 27 if we choose to use only lower case character set and the space.
- depth(T) is very small, usually not more than 10.

The typical value of ϕ and the attempts to control it will be discussed in 4.1 and 4.2.

Chapter 4:
Constructing PHT Models for English: Implementation and Results

4.1 Implementation Issues

Four algorithms for PHT construction were presented in sections 3.1-3.3: The greedy algorithm, the simple dynamic programming, the pseudo-binary tree variation and the hybrid algorithm. The implementation of those algorithms for the construction of small and informative PHT involves considerations of some other practical issues.
The actual numerical values we used for the parameters that are described in these sections appear later in section 4.2.
The algorithms were programmed in C language, using the Borland C++ compiler. The overall code size for all four algorithms, which where compiled together in one program, is about 3500 lines. We used 486DX2 Intel machine with 16M bytes memory for testing the algorithms and constructing the PHT models for English.

Redundant information in the model
Our general goal is building a small, yet informative, model for English. The model is used for string probability evaluations within a character oriented applications. The size of the model has a major effect on the application speed and total size, and so removing non informative elements from the model is an important goal. There are two kinds of redundant information in the PHT we should consider:
1. Non informative PHT nodes.
2. Non informative probabilities within a node.

The algorithms that we presented are based on the selection of a 'heavy' subtree from a given weighted tree. Each selection of a node from the weighted tree indicates that a node in the PHT should be

"split", that is, |σ| nodes should be added to the PHT. Each node in the weighted subtree (which we maximized in sections 3.1-3.3) is practically presented in the target PHT by the corresponding node plus the set of all its |σ| children (see section 3.2).

Example:

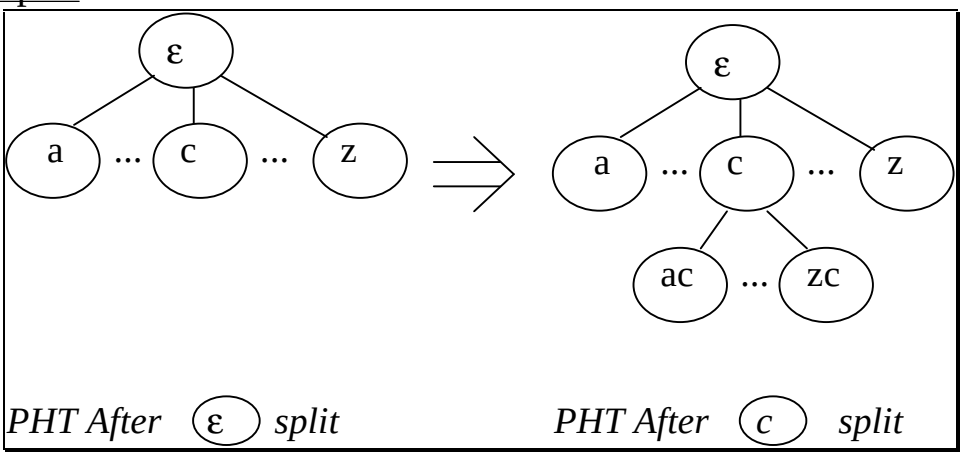


Figure 4.1: A complete split of a node in an English PHT. The node associated with the string "c" is fully "spliced" into |σ| node. Note that some of these new nodes may contain redundant information.

Although we search for the most informative splits, the information gain might mostly come from only part of the new nodes in the PHT, while others may be almost non informative and redundant. This is especially true when we split nodes that match long histories.
In order to overcome this redundancy we make two changes in our problem definition:

- We enlarge the family of accepted models. We include **PHT-s with non equal number of children per node**. If we have a node v which is partially spliced (that is, $0 < |children(v)| < |\sigma|$), then we add an extra child. This child is a *compound node* which keeps the probabilities for all the "missing" children.
- We change the weight function of the nodes. We define the node weight to be the **cross entropy gain of adding this single** node to a tree. This replaces the entropy gain achieved while splitting the node to all its |σ| children. The reduction scheme from theorem 3.2

should be respectively changed: one node in the weighted tree represents now one node in the target PHT, instead of $|\sigma|$ children.

Example:

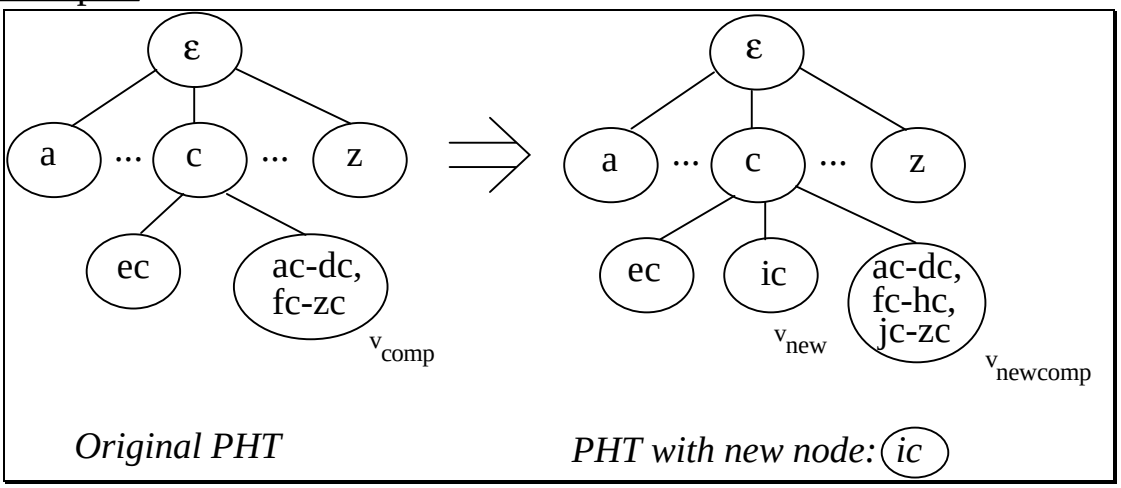


*Figure 4.2: A partial split of a node in an English PHT. On the left: the node associated with the string "c" is partially spliced. It has two children. One is associated with the string "ec". The other is a compound node which is associated with all other "*c" strings.*
On the right: a new child "ic" is added under "c" node. Notice that this action also changes the strings associated with the compound node and its entries probabilities.
The limited number of children and the gradual addition of nodes to the tree reduce the number of redundant nodes.

The new weight function is not well defined since the entropy gain of adding one node to a tree depends on the brothers of the new node. When we use the revised greedy algorithm (described hereafter) to create a PHT the nodes are added to the tree one at a time. We used the temporary tree (which exist at the moment a new node is added by the algorithm) to define the weight for the new node. The weight is defined precisely to be the cross entropy gain for the insertion of the new node to this tree.
In order to simplify the mathematical expression of the weight function we first have to expand the definition of C(v,c) from the beginning of section 3.1 to the new type of compound node v:

$$C_T(v,c) = C(S(father_T(v)) \cdot c) - \sum_{u \in brothers_T(v)} C(S(u) \cdot c)$$

we also expand the definitions for the compound node:

$$C_T(v) \equiv \sum_{c \in \sigma} C_T(v,c), \quad \overline{P}_{T,v}(c) \equiv \frac{C_T(v,c)}{C_T(v)} \quad \text{and} \quad P_T(v,c) \equiv \frac{C_T(v,c)}{R}$$

using the new expansion for C(v,c).

Notice that the expansion for C(v,c) follows the original meaning. C(v,c) is the number of times $P_v(c)$ was used in the process of evaluating $P(X|M_T)$ (see section 3.1).

We also keep the definition of the probability functions set $\Re$:

$$P_v(c) = \overline{P}_{T,v}(c) = \frac{C_T(v,c)}{C_T(v)}$$

which is the distribution being used for evaluations of strings using the compound node v.

Let v_{new} be a new node which is currently added to the tree T, as a child of $\overline{v}$. Assume there exist an original compound child v_{comp} of $\overline{v}$, and a compound node $v_{newcomp}$ after the insertion of v_{new}. We follow the calculations from section 3.1 and we calculate the cross entropy gain for adding the new node v_{new} to T:

$$H(X;M_T) - H(X;M_{T'}) =$$

$$\sum_{\substack{v \in leaves(T') \\ c \in \sigma}} P_{T'}(v,c) \log(\overline{P}_{T',v}(c)) - \sum_{\substack{v \in leaves(T) \\ c \in \sigma}} P_T(v,c) \log(\overline{P}_{T,v}(c)) =$$

$$\sum_{c \in \sigma} [P_{T'}(v_{new},c) \log(\overline{P}_{T',v_{new}}(c)) + P_{T'}(v_{newcomp},c) \log(\overline{P}_{T',v_{newcomp}}(c)) -$$

$$P_T(v_{comp},c) \log(\overline{P}_{T,v_{comp}}(c))]$$

Denoting $e_T(v,c) = C_T(v,c) \log \frac{C_T(v,c)}{C_T(v)}$ we get:

$$\Delta_H = \frac{1}{R} \sum_{c \in \sigma} [e_{T'}(v_{new},c) + e_{T'}(v_{newcomp},c) - e_T(v_{comp},c)]$$

If v_{new} is the first child of $\bar{v}$ then there is no original compound node and we get:

$$\Delta_H = \frac{1}{R}\sum_{c\in\sigma}[e_{T'}(v_{new},c) + e_{T'}(v_{newcomp},c) - e_T(\bar{v},c)]$$

If the insertion of v_{new} completes the split of $\bar{v}$ to all $|\sigma|$ characters then there is no compound node after the insertion and we get:

$$\Delta_H = \frac{1}{R}\sum_{c\in\sigma}[e_{T'}(v_{new},c) - e_T(v_{comp},c)] = 0$$

This is since the information left in the compound node is exactly the information of v_{new}. This insertion practically never happens, and so the maximum number of children is $|\sigma|$.

The choice of the new weight function is heuristic and it shows good results. It can be justified by the fact that the greedy algorithm gives very good results, and hence, the brothers of the new node (when it is added) are quite sure to be part of the optimized subtree as well.
The only reason for the expansion of the models family and the change in the weight function is the attempt to reduce redundant nodes in the target PHT. If one is willing to use only PHT-s with exactly zero or $|\sigma|$ children per node then the original weight function is correct and precise.

Another aspect influencing the 'size to information-content' ratio is the redundancy within a PHT node. Each node in the PHT normally holds a list of $|\sigma|$ values describing the probability of each next character provided that the suffix of the given sequence is the string associated with the node.

Example:

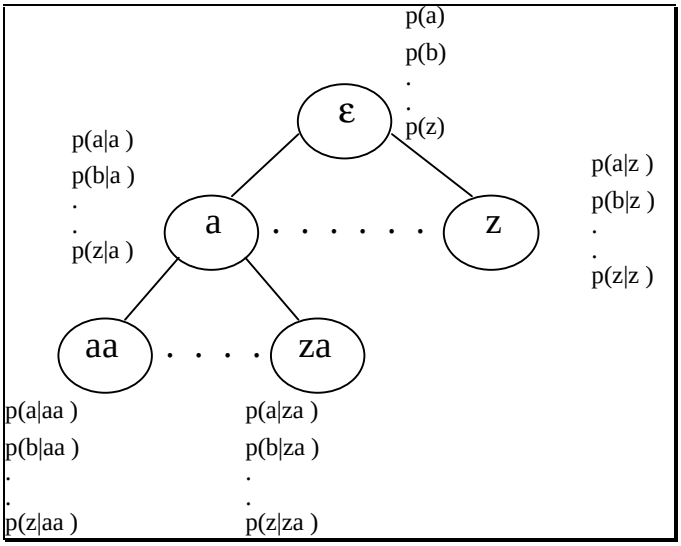


Figure 4.3: Entries of a node in an English PHT. Each node contains $|\sigma|$ entries. Some of these entries may contain zero or near zero probabilities.

The next character probability is not always important. Some of the conditional probabilities are zero or very small and represent strings which do not (or rarely) appear in the language. These probabilities can be removed from PHT, indicating their zero value by their absence.

We use a cutoff parameter λ to remove these redundant small values from the PHT. When using the model in an application for evaluating strings probability we assigned the constant $\mu_{\mathbf{prob}}$, some small positive probability for all the missing entries in the nodes (i.e. for all entrieswith probability $0 \le P \le \lambda$). This has a smoothing effect and helps us to avoid the immediate rejection of rare strings.

CPU and memory needs for PHT construction

Although the construction of the PHT is done just once, cpu time and memory consumption of the algorithms should be reasonable. The algorithms were implemented such that they will run for not more than a few hours on a 486DX2 Intel machine with 16M bytes memory.

In order to overcome the above redundancies, and to control the speed and memory limitations we implemented and tested two slightly modified versions of the algorithms presented in sections 3.2 and 3.3:

Revised greedy algorithm

The implemented version for the greedy algorithm from section 3.2 does not make a complete split of a PHT node to all its children. Instead, each child is added separately as described above. At each step the algorithm looks for the most informative single child to be added using the new expanded definition of the weight function. When the child is added the cutoff parameter λ is used to remove redundant low probabilities from the child's entries. The child's weight is set to be the information gain achieved by its insertion to the tree. The compound brother of the new node is being updated.

In order to speed up the algorithm it was implemented using the array A[v], as suggested in the proof for the algorithm's computational complexity (for more details see section 3.2).

The original tree from which the greedy algorithm extracts the PHT is the full Markov tree of order L. Since this tree might be huge we practically never build all of it. Instead, when a new node is added, the algorithm scans the training set for the probabilities of all its children, and so the potentially needed nodes are always available. The simple scanning can be replaced with some hashing or indexing algorithms which will speed it up.

'Best-PHT' algorithm

We have implemented a very fast optimizing algorithm for PHT maximization. The algorithm Best-PHT uses a mixed technique from the Pseudo- Binary algorithm and the hybrid algorithm. It has four steps:

- step 1 - A large PHT is created using the **revised greedy algorithm**. This PHT substitutes the full Markov tree from which we are going to prune our target PHT. The size of this PHT should be large enough to promise that the optimal target PHT of the desired size will be contained in it. Since this large PHT is much smaller than the huge full tree, the optimization stages are faster,

and memory needs are limited. The revised greedy algorithm is used also to define the weight for each node.

- step 2 - The large PHT is converted by the function make_bin to the corresponding **pseudo-binary tree**.
- step 3 - The **hybrid technique** is used to search for a binary subtree with the desired number of nodes.
- step 4 - A general subtree is constructed from the binary subtree.

Controlling the duration of Step 3

In step 3 of Best-PHT algorithm we are using the hybrid algorithm. At the end of section 3.3 we saw that ϕ, the upper bound on the size of Φ, is a key parameter in the computational complexity bound for the algorithm. The hybrid algorithm tends toward exhaustive optimization when the number of 'non monotone elements' in the sequences $d_{k,1}...d_{k,size(v_k)}$ is large. In many cases the next item in the list may deviate just slightly from the monotone sequence. In order to speed up the algorithm we allow an element to slightly deviate from the monotone sequence and still be treated as part of it. This decreases ϕ but gives a solution which is not guarantied to be optimal. The deviation is controlled by a parameter ε_{dev}. As will be described in section 4.2, we found that even a very small value reduces the running time of the algorithm dramatically with almost no loss in the final subtree weight.

4.2 Test Models for English and Results

In order to test both PHT size and PHT construction method we constructed and tested a set of models. There are some common parameters to these models:

- They are all constructed from the same training set which is a large (1.5M characters) collection of texts from a variety of modern English sources, such as a some items from a general encyclopedia, some New York Times articles and a set of American jokes.

- They are all based on the same 30 characters alphabet containing 26 letters and 4 other characters: space, comma, dot, and tag ('). All upper case letters in the training set are converted to lower case. All other characters are ignored.
- Each node can contain up to 30 entries for the 'next character' distribution information. However, we use the cutoff parameter λ**=0.002** to remove from the XPHT-s all entries with conditional probabilities lower than λ. We used the parameter μ_{prob}**=0.0001** for all probabilities of removed entries.
- All the PHT-s (although their source and construction may differ) were transformed into the same data structure. This structure (called XPHT) will be described in section 5.2. The files describing the trees contain the additional data for the XPHT format, that is, an extra pointer associated with each probability.

We constructed three types of models:

- A set of PHT-s for the full Markov models of different orders. We have 3 models named f1, f2, and f3. **f1** is the zero-order Markov model describing the simple distribution of English letters (and other alphabet characters which we used). **f2** is the first order Markov model. It contains all probabilities of the type $p(c_1|c_2)$ and f1 probabilities as well. **f3** is the second order Markov model and contains all probabilities of the type $p(c_1c_2|c_3)$ ('trigrams') plus f1 and f2 probabilities.
- A set of PHT-s constructed by the revised greedy algorithm presented in section 4.1. This algorithm adds one node per iteration to the PHT. We build four models with growing sizes. The first two are called **n1** and **n2** have roughly the sizes of f2 and f3 respectively. **n3** and **n4** are larger. The maximum depth L of the PHT-s is 10.
- Three PHT-s called h1, h2, and h3 constructed with the Best-PHT algorithm from section 4.1. **h1**, **h2** and **h3** have roughly the sizes of n1, n2 and n3 respectively. The size of the original PHT built by the first greedy step of the Best-PHT algorithm is 4300 nodes (40010 entries). We found out that using a deviation parameter ε_{dev}=0 for

constructing the PHT-s is computationally feasible, since the pseudo-binary technique is only quadratic in the tree size. This means that all h* PHT-s are optimal with respect to the weight function, with the exception of redundant entries which were removed using λ.

Following is a table of all test models. For each model we give the number of nodes, the number of entries (probabilities), the file size in bytes, and the entropy of the model. The size of XPHT file is the number of model entries multiplied by the size of one entry plus the size of a small header in the beginning of file. We used a fixed number of four bytes per XPHT entry: one byte for the next character, one byte for the minus log of the probability and two bytes for the edge information. This file should be present in memory when the application is resolving ambiguous text (see section 5.1 and 5.4), and its size is a key parameter for a small computer implementation.

model	type	nodes	entries	file size	entropy
f1	full Markov	1	30	132	4.213
f2	full Markov	31	760	3116	3.456
f3	full Markov	776	7594	30388	2.841
n1	r. greedy alg.	41	753	3024	3.342
n2	r. greedy alg.	570	7603	30424	2.665
n3	r. greedy alg.	1792	19744	78988	2.337
n4	r. greedy alg.	4300	40010	160052	2.117
h1	Best PHT alg.	42	753	3024	3.341
h2	Best PHT alg.	568	7606	30436	2.664
h3	Best PHT alg.	1802	19743	78984	2.334

Table 4.1: PHT based models of different size and class. f1-f3 are full Markov models of the 0 to 2 order. n1-n4 are n-gram models with variable memory length built by the greedy algorithm. h1-h3 are n-gram models which were built using the Best-PHT algorithm.

Observing these models we can derive some conclusions:

- The information content of the models (expressed in a logarithmic entropy scale) is decreasing when the entries number increase. This is also shown in the next graph, which plots the entropy versus the number of entries for the greedy algorithm. It is clear that most of the information is achieved using a small number of entries, while reducing the entropy becomes harder when the size of XPHT increases:

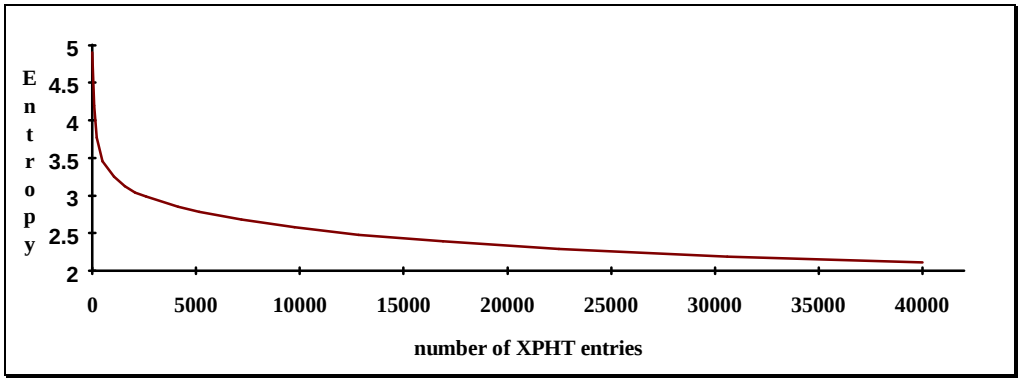


Figure 4.4: The decrease in entropy of the XPHT while it is being grown by the revised greedy algorithm.

- The models that were created using the revised greedy or Best-PHT algorithms are much more informative than the full Markov models with the same size.
- The Best-PHT algorithm showed no significant superiority over the revised greedy algorithm. Nevertheless, this mild entropy difference is not so easy to achieve when the XPHT-s are already rather large. We found out, for example, that an XPHT with the same entropy of h3 can be built using the revised greedy algorithm, but will be larger by about 200 entries (about 1% of the original size).

In addition we tested the timing for our algorithms. We based all our tests on a 4300 nodes PHT which was built using the greedy algorithm. This PHT construction took few hours but could be easily accelerated using a simple hashing or indexing techniques. Our main interest was the optimization algorithm feasibility and speed.
We tested inferior versions of Best-PHT in order to find the contribution of each component in the algorithm to the overall speed.

We measured the overall time needed to select the 1000 best nodes from a 4300 nodes PHT by some versions of the algorithm. The test was performed on a 486DX2 Intel machine with 16M bytes memory.

- Using Best-PHT (that is, combining the hybrid technique and the pseudo-binary technique) takes 04:20 minutes.
- Using Best-PHT **without** the hybrid technique in step 3. This means that only the pseudo-binary technique was used, and all combination at each binary node were tested. It takes 11:05 minutes to complete the nodes selection using this version. Comparing this result to the previous one justifies the use of the hybrid technique. Although the pseudo-binary algorithm is quadratic (in the PHT size) we can speed up the optimization combining it with the hybrid technique.
- Using Best-PHT **without** the pseudo-binary technique. This was done by skipping steps 2 and 4 in the algorithm (see section 4.1) keeping only the hybrid technique in step 3. As expected, the performance degrades dramatically. We found out that for $\varepsilon_{dev}=0$ the algorithm takes about 30 hours. However, for error parameter as small as $\varepsilon_{dev}=0.000005$, the algorithm takes about 5 hours.
- Using the simple dynamic programming without both the hybrid technique and the pseudo-binary technique was found completely non feasible.

Chapter 5:
Algorithms for Using PHT in Recognition Tasks

In the previous chapter we discussed the construction of a PHT model for English of a desired size. In this chapter we will show how this model can be used in an algorithm which receives an ambiguous English text and resolves this ambiguity in an optimal way with respect to the PHT. This algorithm was implemented and used successfully in two practical applications which run on a personal computer with limited resources. We will analyze the algorithm's features and present test results for its performance.

5.1 Ambiguous Text Resolution (ATR) Problem

'Recognition task' is a general name for a large set of problems. What characterizes these problems is the attempt to translate (under some correctness criteria) original representation of given data to a different, new, more 'meaningful' representation. This could be the translation of a human face photo to the person name, the translation of handwritten characters to their ASCII codes or the translation of spoken words to a text format.
In this work we are focusing on one type of recognition task. We are interested in translating sequential data which consist of basic elements presented one after the other (e.g., handwritten characters, or human voice which is built of syllables). Moreover, we are interested in recognition methods which use a statistical approach. This kind of methods usually involves the evaluation of two kind of probabilities:

- Probabilities which are related to the translation process of a single element. These probabilities usually describe the distance (in a relevant metric) between the given data and some prototype of the data which is part of the recognition mechanism. This prototype is associated to some element in the new representation language. For each element x in the original sequence we evaluate the probability p(x|y). This describes the chance that element x is associated with y (from the new representation). A common hypothesis (which is

convenient but not always true) is that this probability p(x|y) does not depend on the other elements in the sequence.
- Probabilities which describe the conditional relationship among the elements in the new representation. This relation usually depends on the relative positions of the elements in the sequence.

The general description of the problem is this:

Problem 5.1 - Ambiguous Text Resolution (ATR):

Let σ and $\overline{\sigma}$ be two alphabets. Given are:

- A sequence of n signs $x_1, x_2, ..., x_n$, $x_i \in \overline{\sigma}$.
- A set of probability distributions $p_i(x|c_j)$ for every i=1..n, $x \in \overline{\sigma}$ and $c_j \in \sigma$. $p_i(x|c_j)$ is the probability that x in the i-th position stands for c_j .
- A set of probability distributions $p(c_j|c_1...c_{j-1})$ for every $j \le n$, $c_1, ..., c_j \in \sigma$.

Find $c_1, ..., c_n$ such that $p(c_1...c_n|x_1...x_n)$ is maximal.

In order to distinguish the input from the output we will call the elements of $\overline{\sigma}$ (the input sequence) **'signs'** and the elements of σ (the output sequence) **'characters'**.

In fact, the signs are not essential part of the input to the problem. What we actually get is just the set of probability distributions $p_i(x|c_j)$.

A common assumption is that $p_i(x|c_j)$ is independent of i and that x depends only on c_j. If we use this assumption then the problem can be solved simply by maximizing $p(c_1...c_n|x_1...x_n)$:

$$\arg\max_{c_1...c_n \in \sigma^n}(p(c_1...c_n|x_1...x_n)) =$$

$$\arg\max_{c_1...c_n \in \sigma^n}(p(x_1...x_n|c_1...c_n)p(c_1...c_n))$$

using the fact that each x_i depends only on c_i we can write:

$$= \arg\max_{c_1...c_n \in \sigma^n}(\prod_{i=1..n} p(x_i|c_i)p(c_i|c_1...c_{i-1}))$$

which takes $o(n \cdot |\sigma|^n)$ operations when done in the straight-forward way.

Since the general problem can involve a very large set of parameters it is often the case that further conditions are assumed. For example, we

can use the limited parameters set of the first order Markov model for σ's language by assuming: $p(c_j|c_1...c_{j-1}) = p(c_j|c_{j-1})$ for each j=1..n. Another assumption can be that for each i only some small number of probabilities $p(x_i|c)$ are non zero. This will limit the overall number of candidate strings in the maximization.

5.2 A Probabilistic Finite Automaton Description for a PHT

Resolving ambiguous text is done by assigning probabilities to the different interpretation of the text, and selecting the one with the highest probability. The PHT can be used directly for probability assignment by the method presented in section 2.5. This means that for each character in the text we should repeat the process of finding the relevant node in the PHT which will give us the best possible estimation of the model for the character appearance after its preceding characters.

There is a way to make the process of nodes finding more efficient. It is based on the representation of a Markov model by a PFA (Probabilistic Finite Automaton) discussed in section 2.3. There is of course the question the target PFA size. Since the parameters set of the PHT is much smaller than the parameters set of the complete Markov model, we are not interested in the naive PFA representation of the complete model, but rather in a representation which keeps the small size of PHT.

Ron, Singer and Tishby have shown that any PHT T=(V,E,S,ℜ) can be represented by PFA with at most depth(T)·|V| nodes such that for every history longer than depth(T)-1 the next character probabilities assigned by the PHT and the PFA are equal [Ron 95].

The construction of such PFA from a given PHT is quite intuitive. The full details are presented in their paper [Ron 95] and we avoid repeating them. The construction can be done using a naive algorithm in time quadratic in the number of the nodes of the PHT, which is usually not too large in the practical cases. Here is a simple example of a PHT and its matching PFA taken from Ron's paper:

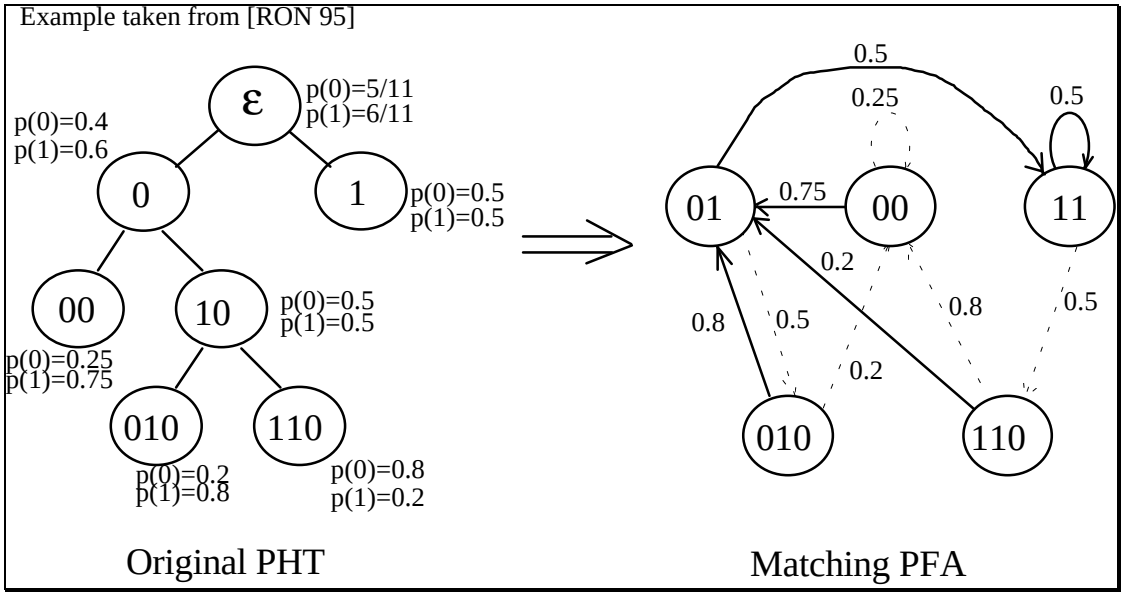


Figure 5.1: A PHT (on the left) and its matching PFA (on the right). The bold edges of the PFA represent walk with character "1" while the dotted edges represent walk with "0". Notice that the leaf "1" in the PHT has been replaced by two nodes "01" and "11" in the PFA. On the other hand, the inner node "0" in the PHT has been removed in the PFA.

The actual representation for the PHT model which we have used for the ATR algorithm (see section 5.4) is a variation on the PFA representation. It is called **XPHT** (short for Extended PHT) and it has the following features:

- The inner nodes are kept in the XPHT although they are removed in the PFA. In real life applications the possibility to assign probabilities for short strings is important. Since we are interested in maintaining the model ability to assign probabilities for strings which are shorter than depth(T) we need the inner nodes of the tree.
- No new nodes are added to the XPHT. When building a PFA from a PHT many nodes might be added to the model. These additional nodes are not guaranteed of the optimization process, and are not promised to be very informative, yet they take place in the model as the original informative nodes. The additional nodes are chosen such that the PFA becomes a full prefix graph, that is, if $c_1c_2...c_{i-1}c_i$ is a string associated to some node v_1 then there exist a node v_2 such that $S(v_2)=c_1c_2...c_{i-1}$ (Note that the original PHT is a full suffix graph). However, our tests results showed that only about 5% of the nodes do not already have prefix nodes in the original PHT.

This minor incompleteness in the XPHT prefix quality hardly effects the model.

- A set of pointers which connect nodes in the XPHT is added in the following way - Let $suffmax_T(s)$ be a function defined for every string s in σ^*. $suffmax_T(s)$ is the maximal suffix of s which has an associated node in T. Let $\delta(v,c)$ be a function defined for every node $v \in V$ and character $c \in \sigma$. $\delta(v,c)$ is the node associated to the string $suffmax_T(S(v) \cdot c)$. For every node $v \in V$ and character $c \in \sigma$ with probability $\geq \lambda$ (see discussion on redundant probabilities in section 4.2) include a pointer from v to $\delta(v,c)$. All near zero probabilities which do not appear in the original PHT are associated with a default pointer to the root node. This default pointer is used after the near zero default probability μ_{prob} has been assigned to a character during the evaluation of string probability. Although the next character probability will be evaluated using zero memory node (the root) the global string probability will hardly be effected since it is already dominated by the near zero value, and is most likely not the desired solution for the ATR problem. This set of pointers does not change the order of the PHT size. This size is counted in ***entries***. **An XPHT entry contains a character, its corresponding probability and a pointer to the next node.** When using an XPHT for assigning string probability the first depth(T)-1 characters are evaluated by the original PHT inner nodes. All subsequent assignments are done by a walk along pointers. Since the pointers to the next node are precalculated, just one step is needed for each character, and no search is performed.

The change from PHT structure to XPHT structure is only done in order to speed up the ATR algorithm. The original PHT structure represents the data accurately, but it slows down the applications. In section 5.4 we will show how the XPHT is used efficiently for assigning probabilities of interpretations for an ambiguous text.

5.3 The Standard Viterbi Algorithm and Shortcuts

In section 5.1 we described the general ATR Problem. In this section we will discuss the simple case of solving the problem using a first order Markov model.
We are trying to find:

$$\arg\max_{c_1 \ldots c_n \in \sigma^n}(p(c_1 \ldots c_n | x_1 \ldots x_n))$$

which is estimated by:

$$\arg\max_{c_1 \ldots c_n \in \sigma^n}[\, p(x_1|c_1)p(c_1) \prod_{i=2..n} p(x_i|c_i)p(c_i|c_{i-1})]$$

using the Markov model, where $p(c_i|c_{i-1})$ is the transition probability and $p(c_i)$ is the stationary probability.
This takes $o(n \cdot |\sigma|^n)$ operations when done in the straightforward way.
In this section we will show that the problem can be solved, or can be approximately solved, using a much faster algorithms.

Define $m=|\sigma|$ and $c_1 \ldots c_m$ to be the alphabet for the rest of this section.
A better view on the problem can achieved by using a *trellis* description. The trellis is a directed graph in which every directed path with n+1 vertices (including the 'start' vertex) describe one unique possible set of $c_{\alpha_1} \ldots c_{\alpha_n}$ in the maximization problem:

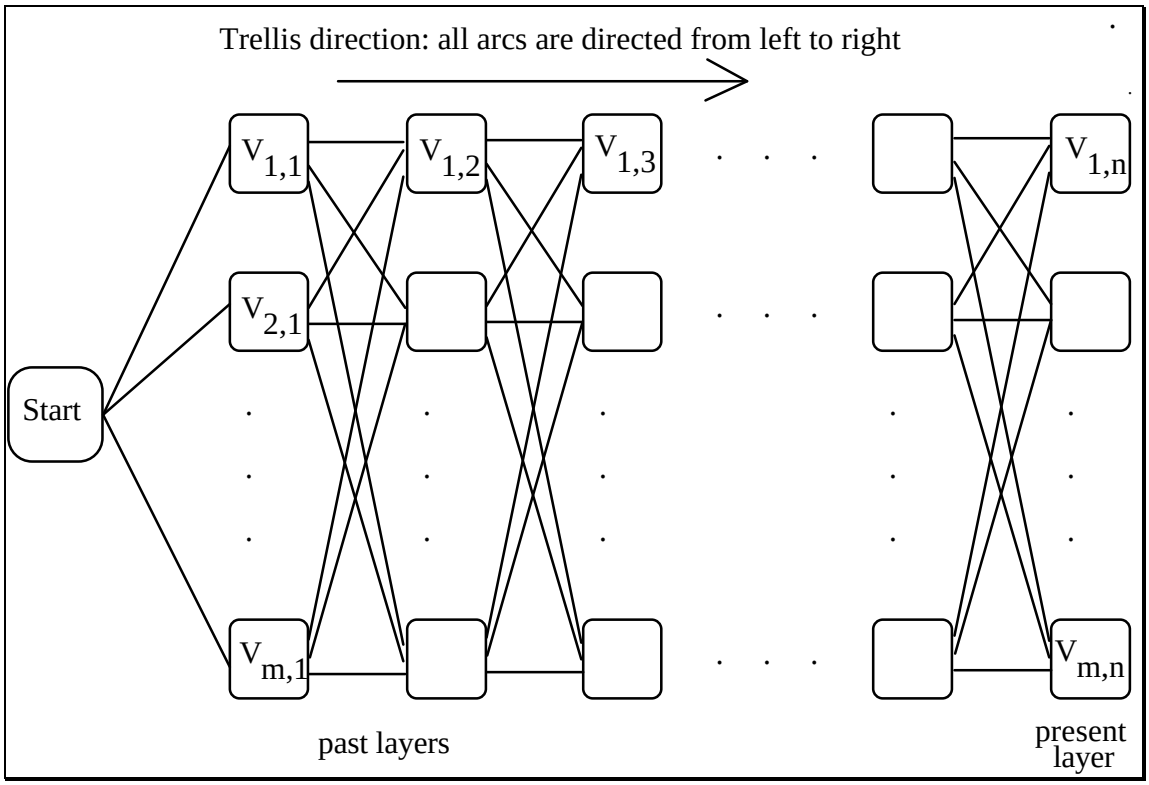


Figure 5.2: A Trellis description of the ATR Problem using first order Markov model. All edges are naturally directed from left to right, although practically it is sometimes more convenient to keep pointers directing from right to left instead. Each layer contains m vertices, and each vertex has m in-edges and m out-edges. Each vertex $V_{i,j}$ *is assigned with the character* c_i *and the probability* $p(x_j|c_i)$. *Each edge from* $V_{i,j}$ *to* $V_{k,j+1}$ *is assigned with the probability* $p(c_k|c_i)$ *taken from the Markov model. Each edge from 'Start' to* $V_{k,1}$ *is assigned with the probability* $p(c_k)$. *In order to calculate the probability of the string* $c_{\alpha_1}...c_{\alpha_n}$ *follow the path starting at 'start', proceeding to* $V_{\alpha_1,1}$, *then to* $V_{\alpha_2,2}$ *etc. proceed until the n+1 vertex* $V_{\alpha_n,n}$ *in the path. Multiply the probabilities assigned to all the vertices and the edges along the path.*

The trellis description leads to a dynamic programming algorithm based on the following scheme:

- Start by solving the maximization problem associated with the single layer (the 'start' and one full layer). That is, for every vertex $V_{i,1}$ compute the probability of the two vertices path starting from

'start' and ending at $V_{i,1}$ by multiplying the probabilities on that path: $p(c_i) \cdot p(x_1|c_i)$.

- Assume that the maximization problem up to the k-1 ($k \le n$) layer in the trellis is already solved and use this solution to compute the solution for the k layer maximization. That is, for every vertex $V_{i,k}$ find a k+1 vertices path starting from 'start' and ending at $V_{i,k}$ which has the highest probability.
- Repeat this process up to the n layer. Choose the n+1 vertices path starting from 'start' which has the highest probability. The characters along this path form an optimal solution to the full problem.

This algorithm is known also as Viterbi algorithm [Viterbi 67]:

Viterbi algorithm

Initialization:

$\forall j=1..m$ let $prob[1,j] \leftarrow p(c_j) \cdot p(x_1|c_j)$

Recursion:

for i=2..n do
    for j=1..m do
        $prob[i, j] \leftarrow \max_{k=1..m}(prob[i-1,k] \cdot p(c_j|c_k) \cdot p(x_i|c_j))$
        $path[i, j] \leftarrow \arg\max_{k=1..m}(prob[i-1,k] \cdot p(c_j|c_k))$

Termination and best path reconstruction:

$best\_prob \leftarrow \max_{j=1..m}(prob[n, j])$

$\alpha[n] \leftarrow \arg\max_{j=1..m}(prob[n, j])$

$i \leftarrow n$

while i>1 do
    $\alpha[i-1] \leftarrow path[i, \alpha[i]]$
    $i \leftarrow i-1$

The set of characters $c_{\alpha[1]} \ldots c_{\alpha[n]}$ is the solution for our problem.

The computational complexity of the Viterby algorithm is dominated by the main recursion which takes $O(n \cdot m^2)$.

The Viterby algorithm can be adapted to an important special case of the ATR problem. This is when only a small number of the probabilities $p(x_i|c_j)$ j=1..m are non zero among all i=1..n. In this case only the nodes with non zero probabilities should appear in the trellis. If there exist a bound $\overline{m} < m = |\sigma|$ for the number of non zero probabilities per layer then the computational complexity of the Viterby algorithm goes down to $O(n \cdot \overline{m}^2)$.
An important generalization is when we want to use the k-th order Markov model rather than the first order model. This can be implemented using the same algorithm with mild changes. The corresponding trellis becomes much larger. At each layer the total number of vertices rises up to m^k. For each combination $d_1 d_2 ... d_k \in \sigma^k$ there exist one unique vertex in every layer. This vertex in layer j is assigned with the probability $p(x_j|d_k)$. The number of edges between two successive layers becomes m^{k+1}. For every two vertices $V_{i,j}$ and $V_{h,j+1}$ such that $V_{i,j}$ is associated with the string $d_1 d_2 ... d_k$ and $V_{h,j+1}$ is associated with the string $d_2 ... d_{k+1}$ there exist an edge from $V_{i,j}$ to $V_{h,j+1}$. This edge is assigned with the probability $p(d_{k+1}|d_1 ... d_k)$. The total work of the Viterbi algorithm using this model becomes $O(n \cdot |\sigma|^{k+1})$.

There are some algorithms in literature (see [Fano 63], [Jelinek 69]) which suggest a maximization method for solving the ATR problem which does not guarantee best solution. The general idea is to stop evaluating the probability of short paths whose chance to develop into the high probability long paths is small. This can reduce the resolving time, and if done carefully, there is not much loss in the quality of the result.
In section 5.4 we will present a variation on the Viterbi algorithm which uses an XPHT model and optimally solves the ATR problem.

5.4 On-Line ATR Algorithm Using a PHT

A batch based application running on a large computer is quite different from a real life, on-line application running on a personal

computer. Some special adaptation and careful design should be done if we want to implement the algorithm for applications like handwriting-based word processor or phone keypad-based communication.
We have to consider some limitations and needs:

- Memory: we are short of memory in general. Contributing to limitation is the total small memory of the computer and the fact that the recognition process is only one component of the main program such as the word processor itself.
- Memory management: we have to take care of memory management, and unused memory should be available instantly for reuse.
- Speed: time is a major factor since the program is part of an on-line interface, and the user is waiting for immediate response.
- On-line behavior: results of the algorithm should be presented to the user as soon as possible. That is, we do not want to wait until all the noisy message is delivered to the application, then let the algorithm analyze it, and just at the end of the process return the algorithm's choice. Rather, we want ambiguity resolution to be done while the message is being delivered. We want the gap between arrival of one input sign and recognition of the corresponding character to be as short as possible.

Following is a simplified version of the implemented algorithm which we have built using the dynamic programming scheme of the Viterbi algorithm (see section 5.3). We focus here on the way the main data structure of the algorithm is built up from the ambiguous input with the assistance of the language model. The full implementation is complex and involves many details, some more technical and others specific for an application.

Main data structure

The implementation is based on a rooted DAG (directed acyclic graph) structure. The DAG is built up layer by layer where each layer represents a set of the XPHT nodes (states). Processing each new character from the input sequence generates a new layer at the '*front'* of the DAG. However, as opposed to the Trellis described in section 5.3 the '*tail'* ('old' layers) of the DAG is gradually disappearing while the front develops. Most of the tail's nodes which describe paths

(strings) with low probabilities are removed, while the rest, which describe the most likely path are collected to form the interpretation of the ambiguous message.

Each node v in the DAG has the following attributes:
char - the character associated with the node.
state - a pointer to the state (node) in the XPHT corresponding to the node (this pointer may change during the algorithm).
prob - minus the logarithm of the probability of the path (possible string) ended by the substring associated with the state.
active - a flag which is true if and only if a state is still informative for the next iteration. A state is still informative if it may be used to evaluate probabilities in the next iteration. Nodes whose active flag is true are called **active** while nodes marked false are **inactive**. The set of active nodes will always form a rooted directed subtree in the front of the DAG (lemma 5.1).
best - a pointer to the previous node on the most probable path ending in v. This pointer is initialized to 'nil' and changes its content during the algorithm's progress. It assumes its final value only when all v's children become inactive.

Each node in the DAG is pointing to all of his **children**. The active part of the DAG is a tree, where the familiar terms of **father** and **child** holds. The ε node is always the ancestor of all the nodes in the DAG.

The input format

Each sign x in the input is represented by a vector of suggestions and their probabilities. Each entry of the vectors has the form (char, prob) where char is a character and prob=p(x|c). We assume that the probabilities in the vector sum up to one.

The language model

The language model used is a XPHT of English. The XPHT was built from a PHT as described in section 5.1.
δ(state, character) is the XPHT pointer function.
p(state, character) is the conditional probability associated with the pointer δ(state, character) in the XPHT.

All probabilities are transformed to minus logarithm scale and are added by the algorithm (instead of multiplying the probabilities). This means that the winning string is the lowest scored one.

Simplified version of the on-line, PHT based ATR Algorithm

Initialization:

Create a DAG D with a single node v.

Set v.char←ε, v.state←ε ,v.prob←1, v.active←true, v.best←nil

Iteration:

Get suggestions vector $\bar{g}$ for the next sign.

Let D^A be the active subgraph in the front of D.

For every suggestion g in $\bar{g}$ do

begin

Let c ← g.char. Let D_c be a copy of D^A.

For every node v in D_c do

begin

v.prob ← v.prob+p(v.state,c)+g.prob

v.state ← δ(v.state,c)

end

For r, the root of D_c, let r.char ← c

end

Create a node v and set:

v.char←ε, v.state←ε ,v.prob←1, v.active←true, v.best←nil.

Update D by adding arcs from v to the roots of each D_c and removing D^A. Note that the new D still contains all the original inactive nodes, with pointers from nodes in each of the D_c-s.

Perform a postorder DFS on the active tree in D. While 'DFSing', for every node v but the root, if v.state=father(v).state then

begin

v.active ← false

if v.prob<father(v).prob or father(v).best=nil then

begin

father(v).prob ← v.prob

father(v).best ← v

end

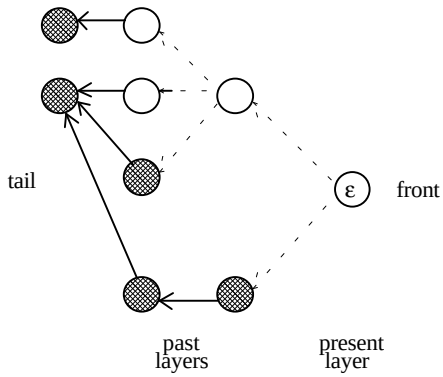
end

remove all inactive nodes which are not connected to some active node by a path of 'best' pointers.

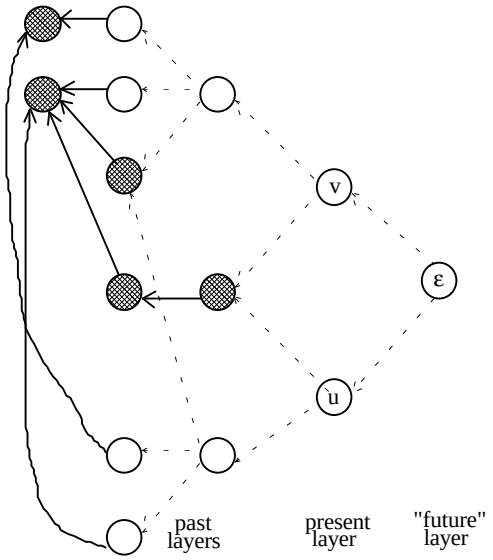
output all characters of the nodes which are inactive and single at their layer. Remove those nodes from D.

Following is an example of one iteration of the algorithm. Empty nodes are 'active' and the crossed nodes are 'inactive'. Dotted arcs are possible paths from the node back to pervious nodes, while full arcs are 'best' pointers which were set to their final value. Notice that the arcs are directed 'backwards' as opposed to the trellis in section 5.3.

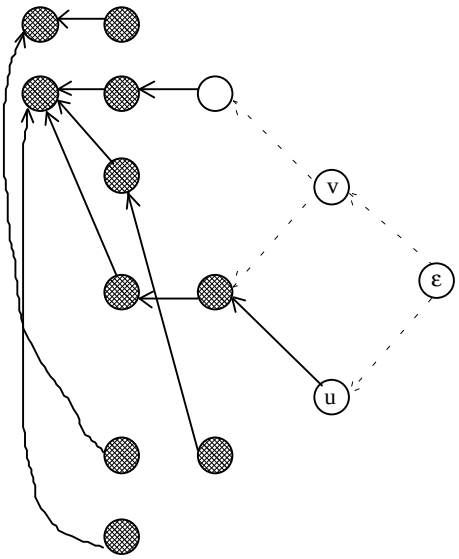
***Step 1**. Original DAG (result of previous iteration of the algorithm).*



***Step 2**. A two characters vector ('v','u') is processed. The original active subgraph is duplicated.*



***Step 3**. During the DFS some nodes become non-active and 'best' edges are updated.*



***Step 4**. The DAG at the end of the iteration. Notice that the dark node is left single at its layer. Its character is output.*

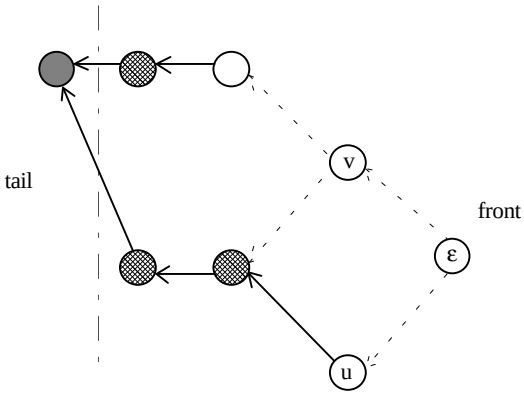


Figure 5.3: One iteration of the ATR Algorithm.

The details of the memory management and DAG structure control are rather tedious and are not presented as part of the algorithm. In general, the DAG structure is kept by updating an edge list for each node. The memory management is done by updating a counter field for each node with the total number of arcs pointing to it. When the count is zeroed the node's memory is freed for reuse.

Lemma 5.1:

1. At the beginning and the end of each iteration of the algorithm the active part of D is isomorphic to a rooted subtree of the XPHT.
2. The size of the active part of D at any time during the iteration is bounded by $N \cdot |\bar{g}|$ ($\bar{g}$ is the suggestions vector for the new sign).

Proof:

At the beginning of the first iteration there is only one node in the DAG. Since the end structure of the DAG at each iteration is the opening structure in the next iteration, it suffices to prove 1 for the end of each iteration.

At the first stage of the iteration, the active part of D is being duplicated for each character of in the vector.

After all duplications D reaches its largest size in the iteration, so, if 1 is proven then 2 follows immediately.

Take some active node v of D, and let v' be a child of v in the active tree. The fields 'state' contain the corresponding nodes in the XPHT: w and w'. We shall prove by induction that w is the father of w'. Assume that w is a father of w' at some stage of the algorithm. This means that $S(w')=hS(w)$ where h is some character and S is the function mapping each node of the XPHT to a string.

Now, v and v' are duplicated for D_c. The state fields of v and v' are updated using δ, that is:

$v.state \leftarrow \delta(v.state,c)=\delta(w,c)=u$, $v'.state \leftarrow \delta(v'.state,c)=\delta(w',c)=u'$.

Since δ is defined such that $S(u)=suffmax(S(w)c)$ and

$S(u')=suffmax(S(w')c)=suffmax(hS(w)c)$

two situations might occur:

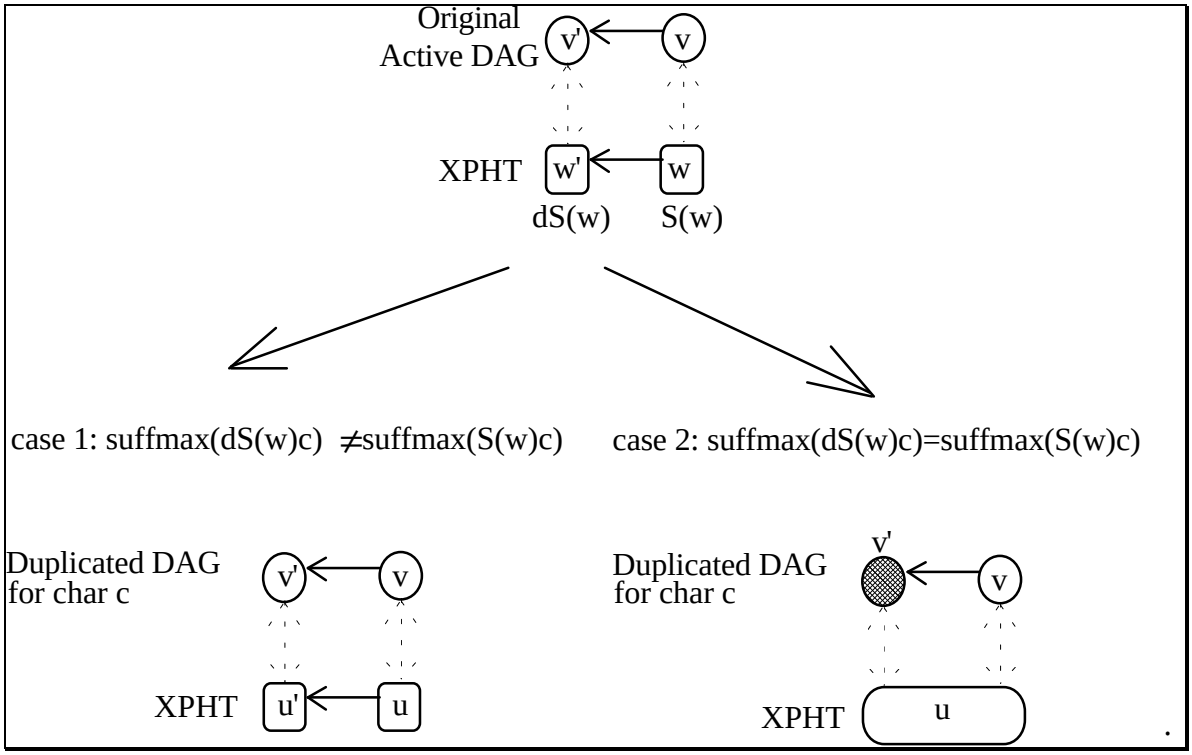


Figure 5.4: father-child relation (solid arrows) in the original active DAG (top) are correlated (dotted arrows) by the field 'state' to the corresponding nodes in the XPHT. After the next character c is added, two cases may occur: In case 1 (bottom right) the correlation still holds, and both nodes remain active. In case 2 (bottom left), the father and the child point to the same node in the XPHT, and the child becomes inactive.

1. If u≠u' then S(u')=hS(u') and u is the father of u' in the XPHT. Hence, the isomorphism is kept in the active part of D_c.
2. If u=u' then the second stage of the algorithm marks v' to be inactive. Moreover, if
   S(u)=suffmax(S(w)c)=suffmax(hS(w)c)=S(u')
   it is clear that for every character k and every string s:
   suffmax(shS(w)c)=suffmax(kshS(w)c).
   This means that all descendants of v' will also be marked inactive and will not be included in the active part of the DAG at the end of the iteration.

It remains to prove the induction claim when v is the new root, r, of D. r is pointing to all the roots of the duplicated D_c-s. Let v be the root of some D_c. r.state:=ε and v.state:=δ(ε,c).

If δ(ε,c)≠ε then surly S(v.state)=c and v.state is the child of r.state.

If, on the other hand, δ(ε,c)=ε then v.state=r.state and v becomes inactive.

Since the XPHT is a tree, and every father-child relation in D^A is maintained by the state pointers, it follows that D^A is isomorphic to a subgraph of the XPHT. Adding the root ensures connectivity.♦

5.5 ATR Algorithm: Features Analysis and Tests Results

In the previous section we presented the dynamic programming algorithm which uses a given XPHT to solve the ATR problem from section 5.1.

A tight analysis of the algorithm's memory consumption and computational complexity depends on the features of the signs generating source and on the dependencies between the signs and the characters. These parameters are particular to the specific application which the algorithm serves, and even to the specific user behavior (type of handwriting, for example). However, general bounds can be derived using lemma 5.1.

In the following subsections we will analyze three main features of the algorithm: memory consumption, speed and on-line behavior. For each feature we will also report a representative experimental test which will show the typical behavior of the algorithm in real applications. Following first is a brief description of the test components.

Algorithms implementation, applications and representative test

The complete PHT based ATR algorithm was programmed on a PC platform in C language, compiled with a Borland C++ compiler. The code size for the on-line algorithm is about 2000 lines and it is optimized carefully for small code size and high speed.

The algorithm was tested in two applications: An on-line handwriting recognition system and a phone keypad-based communication system.

The applications were tested on a 386, 33Mhz Intel processor based board with 4M bytes memory which is currently considered a relatively low cost and slow machine. The on-line handwriting recognition system is a commercial product and our algorithm is only a small portion of it. The keypad-based system is an artificially designed system where the keys of the standard US telephone are used to send alphabetic text over the phone lines. The **full description** of the applications and their **accuracy test results** is presented in **chapter 6**.
The input vectors of suggestions constraints and the test set were specific for each application:

- Handwriting recognition system - The input for the algorithm was a variable size suggestions vectors of at most 6 characters each (2.12 characters on the evarage). Each vector was produced by comparing one sign to a set of prototypes. If the prototype's distance from the sign (measured using a special norm) was small enough then the character associated with the prototype was added to the vector. The handwriting recognition system was tested on a variety of writers and texts (about 24000 characters which were written by 24 writers).
- Phone keypad-based communication application - Each vector contained 3 suggestions which are the letters associated with one key in the standard US phone keypad. We used the same coding conventions for the keypad used by Skiena (see more details in section 6.1 and [Skiena 94]). The results given below are based on testing 1.1M character file containing a selection of Clinton speeches from 1993-94. The Speeches can be found in the Library of Congress, and are available through Internet.

For both applications we used **the same 4300 states XPHT**, with about 40000 entries (an entry in the XPHT is a pointer and a corresponding transition probability. For more details on the test XPHT see the description of model f4 in section 4.2). Following is an **analysis** of the algorithm's main features together **with experimental results** which represent the real behavior of the algorithm for the two application.

Memory consumption

Several general observations concerning the memory consumption of the algorithm can be made independently of the specific application in which the algorithm was used:

- The size of each active node in the DAG depends only on the alphabet size for it holds a list of edges pointing to other nodes. If the vector size is a smaller value limited by $\overline{m}$, as in the case of the phone keypad coding, then the list size is at most $\overline{m}$. The winning string is built by "walking back" on the DAG and there is no need to keep more than one character per node.
- The number of active nodes is bounded by N times the maximal size of the suggestion vector. This is stated in part 2 of lemma 5.1.
- The size of each inactive node is O(1) and does not deepened on the alphabet size or vector size. This is since the node contains one character and points backwards to only one possible path (older node).
- The number of inactive nodes is theoretically not bounded. This is an inherent result of their task - to keep the strings which their destiny as chosen or non chosen might still be changed by the new signs. A simple example is:

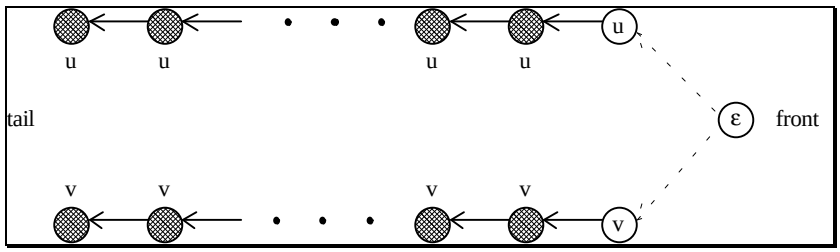


Figure 5.5: An unbounded size DAG.

The same sign is observed over and over again with the ambiguity of being either 'u' or 'v' with the same a-priori probability. The XPHT of the example uses only one character history, and the 'uu' and 'vv' nodes always win. The double path DAG does not return any result since even the oldest (leftmost) 'u' and 'v' are not sure yet. One definitely observed 'u' (or 'v') will collapse the double path leaving only the 'uu...uuu' string as a sure result.
This theoretical lack of bound does not seems to effect the functioning of the algorithm within the tested applications. This is since the problematic situations, like the one described above, are rare in human language. When the space available for active and non active nodes is filled the algorithm can make some heuristic choices for the "old" characters, and cut off the long DAG's tail. This will not effect the average accuracy of ambiguity resolving since it happens rarely.

Experimental results:

We have tested both applications for their memory consumption (the portion of memory that was devoted to our algorithm) measured in DAG nodes. We count the active and non active nodes together.

The phone keypad decoding application used an average number of 70 nodes. Our test shows a maximal use of about 150 nodes.

The handwriting recognition was more demanding in the maximal nodes used. We found a DAG of about 1200 nodes in one case. This is probably since the vector size can grow up to 6 and a sequence of some large vectors can enlarge the DAG. However, the average use of nodes was about 70, the same as in the phone keypad decoding case.

Speed

The overall number of operations per iteration of the algorithm will be evaluated separately for handling the nodes which are active and inactive at the beginning of the iteration:

- **Active nodes** - The number of operations per iteration performed by the algorithm while handling the active nodes is proportional to the maximal number of active nodes that appear in the DAG during the iteration. This number is bounded according to lemma 5.1 by the size of the XPHT times the maximal size of vector. The portion of DAG which is active in the beginning of the iteration is duplicated for each suggested character in the vector, and then the new active structure is searched again to mark new inactive nodes. So, each node in the duplicated structure is reached twice, once when it is being created and once when it is being tested for inactivation. This is true since the duplicated structure is a tree and the DFS reaches each node just once.
- **Inactive nodes** - In order to analyze the work for the inactive nodes in the DAG we should consider the overall work for these nodes during a large set of iterations. When a node becomes inactive it is not reached any more during the two DFS-s of the main iteration. It is actually handled just once more - when the time comes to erase it from the DAG. We assume that the DAG's size stabilized after some iterations and it does not grow endlessly. This can not be proved, however, it was found true in the cases we tested. **If** the size of the DAG is more or less stable after some iterations then the average number of nodes added to the DAG is

equal to the average number of inactive nodes removed from it. This means that on the average the number of operations per iteration for handling the inactive nodes is proportional to the number of the new nodes. The number of new nodes is bounded by the size of the XPHT times the maximal size of vector. This follows from lemma 5.1 since all the new nodes are copies of nodes which were active at the beginning of the iteration.

The conclusion of this disscussion is that if the size of the DAG is stable during the resolution process then the work for the inactive nodes is, on the average, bounded by the size of the XPHT times the maximal size of the input vector.

Experimental results:

We tested both applications for their speed. Our measure is the number of signs resolved by the algorithm per second.

In the phone keypad decoding application the algorithm is the main time consumer (some insignificant portion of time is devoted to files input/output and user interface). The algorithm decoding rate was about 30 decoded characters per second.

The handwriting recognition application devotes only a small portion (less than 13%) of the overall recognition time to the ATR algorithm. The rest of the time is devoted to the 'visual' analysis of the signs, to the creation of the suggestions vectors, and to input and output procedures. The average decoding speed of the algorithm itself was about 33 decoded characters per second (this number is the cumulative time spent in the ATR procedure divided by the number of recognized signs). This means that with the tested XPHT, the algorithm is suitable for on-line handwriting recognition performed on a small computer, since the user writing rate is normally slower than the recognition.

On-line behavior

The number of signs that are inserted to the algorithm before the first sign is resolved has a major impact on the usefulness of the algorithm in an on-line application. This parameter is bounded by the maximal number of layers (both active and non active) in the DAG during the period between the sign insertion and resolution. The tail of the DAG is being used to give a most likely result when there is no ambiguity left there, that is, when a tail's layer is made of only one inactive node.

When this happens the sign is resolved and the tail is being cut off from the DAG.

Experimental results:
An average of 10 layers DAG was measured in the phone keypad text decoding test. This means that an average delay of about 2 English words exist between character encoding and decoding. As mentioned before, the number of layers can grow indefinitely in the worst case. However, the maximum number of layers reached in the test was 30.
The handwriting recognition application used 11 layers DAG on the average. A maximum number of 34 layers in the DAG was reached during the test.

The following table summarizes all the parameters that were discussed above for both the phone keypad text decoding and handwriting recognition:

		phone keypad text decoding	**handwriting recognition**
Tree size (entries)		40000	40000
Test sequence size (signs)		1.1M	24000
Input Vector Size (chars/sign)	**Average**	3	2.12
	Max	3	6
Memory: DAG size (nodes)	**Average**	70	69
	Max	150	1200
Resolving Speed (signs/sec)	**Average**	30	32
Resolving Delay (signs)	**Average**	10	11
	Max	30	34

Table 5.1: Features of on-line, PHT based, ATR Algorithm - applications test results.

The accuracy in resolving signs by both application is highly effected by the size and features of the XPHT being used by the algorithm. Test results of the accuracy with respect to a set of different XPHT-s are given at chapter 6.

Chapter 6:
Results of PHT Building and ATR Algorithms: Two Applications

Chapter 3 presented methods for constructing an informative model of language with a desired size. We have used those methods for constructing a set of models with growing sizes (chapter 4). Chapter 5 described an efficient procedure that uses such a model for resolving ambiguity of noisy or ambiguous text. In this chapter we will present two working applications that use this procedure. We will describe each application and list test results which demonstrate the correlation between model size and application error rates.

6.1 Phone Keypad Interfaced Communication

Lately there is growing interest in applications which use the combination of computer and telecommunication. While duplex telecommunication had already reached virtually every home in the western world this is not the case, yet, with computers. The lake of computer or some terminal device in most houses which do have a telephone line limits the number of users for such applications dramatically. An interesting attempt has been done by some systems aiming to use standard telephone keypad as a simple terminal. This is implemented trivially for limited multiple choice situation but is much harder for complex data coding. On the keypad of most U.S. telephone sets we find three characters written near each of the 2-9 keys. An intuitive user interface for sending a textual massage by a phone keypad can be typing it, characters by character, using the overloaded keys. The entropy of English was estimated by Shannon [Shannon 51] to be between 0.6 to 1.3 bits per symbol. The maximum information rate that can be transmitted using 10 phone dial of phone keypad characters 0-9 is $\log_2(10) \approx 3.32$ bits per symbol. The standard assignment of the letters on the keys is not optimal so the actual

transmission rate when coding an English massage is less then 3.32. Still, it seem that a good model for English can be sufficient to resolve the overloaded digits received through this channel.
A reliable system which can convert the keypad digits back to English text can be used for a variety of applications, mainly when combined with some voice responding system or text-to-speech system. Such a combination can be use for:

- Direct communication with a far computer for information retrieval, goods order, etc. without human intervention on the computer side.
- E-mail sending and receiving through telephone. The far side reads the incoming mail to the user using a text to speech synthesizer.
- Communication with hearing-impaired. The far side must have a computer that resolves the massage ambiguity and uses a text to speech system for responding.

Some works and patents which attempt to solve the problem using statistical methods and dictionaries can be found in literature. An impressive work in this field was done by Rau and Skiena [Skiena 94]. Their system is computationally very heavy and ambiguity resolving is a complex process involving some layers:

1. Blanks recognition using third order Markov model for English.
2. Words match: this is done in two ways - hashing for words in English dictionary, or (for unknown words) scoring candidate words using a trigram model (second order Markov model).
3. Sentence disambiguation resolving which is done by using word-pair frequencies in English and grammatical constraints.

Their system has the following features:

- Complex algorithm combining a set of different mechanisms.
- The model is based on a very large set of parameters.
- The resolving is not on-line, that is, it is done after each complete sentence ends.
- The results are very accurate in general (99.04% correct characters for the test set of Clinton speeches, 95.20% for another test set of Shakespeare works). The accuracy in correctly decoding unknown words (which is done using the Viterbi algorithm with a second

order full Markov model) is only 87.28% of the total number of characters in the Clinton speeches test set.

We have addressed the same problem using the variable memory length PHT model of chapters 2.5-4 and the resolving ATR algorithm suggested in section 5.4. We used the same coding conventions used by Skiena:

- The digits 2-9 represent the characters a-y, three characters per digit, with the 'q' missing from the sequence (this is the standard phone keypad convention).
- The '*' sign is used for the blank, 'q' and 'z'.

Our system has the following features:

- The model is based on a small set of parameters. For any desired model size an optimal (or near optimal) usage of parameters space is achieved.
- The system can be implemented on a small, slow and limited-memory computer and works fast enough for convenient on-line application.
- The resolving algorithm has a desirable on-line behavior. There is a relatively small gap between encoded and resolved character (about two words).
- The results are accurate with respect to the model size (95.5% for the set of Clinton speeches, 91.2% for Shakespeare, using the n4 PHT model).

The following table presents system accuracy with respect to model size. The system was tested using the Clinton speeches file of about 1.1M characters, and a portion of Shakespeare works of the same size. The file was first converted to lower case characters only, when all separators (commas, dots etc.) converted to blanks and all other special characters removed. The result file was converted to the digits 2-8 and '*' character as describe above. This file was decoded using our system and the result was compared to the original lower case file.

model	nodes	entries	entropy	accuracy (%)
uniform distribution	0	0	4.906	33.3%
f1	1	30	4.213	64.6%
f2	31	760	3.456	76.3%
f3	776	7594	2.841	88.0%
n1	41	753	3.342	80.0%
n2	570	7600	2.665	90.4%
n3	1792	19744	2.337	94.2%
n4	**4300**	**40010**	**2.117**	**95.5%**
h1	42	753	3.341	80.0%
h2	568	7606	2.664	90.6%
h3	1802	19743	2.334	94.2%

Table 6.1: The accuracy of PHT based telephone code reconstruction system with respect to parameters number and model construction technique. (Test set: Clinton Speeches)

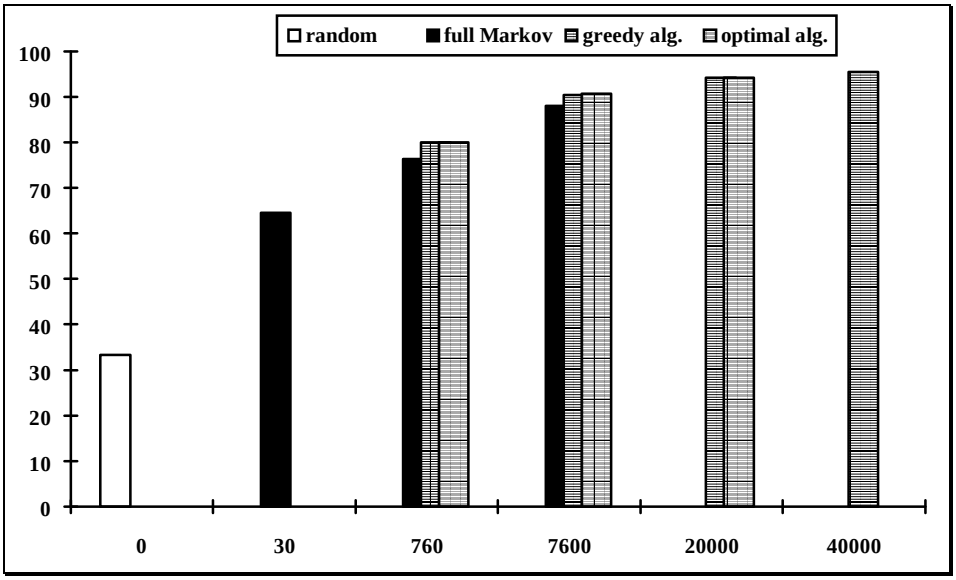


Figure 6.1: System accuracy with respect to parameters number. The four types of bars represent the four different algorithms used for PHT construction. Each set of attached bars represent some XPHT-s with same sizes.

Here is an example from the Clinton speeches file decoded by the phone keypad conventions and encoded by our system using the n4 PHT with 40000 parameters. All mistakes are followed with the correct word in parenthesis:

"...in this kind of en**t**ironment [environment] it is understandable that change would become the watchword of this time what is the **ac**taly**p**t [catalyst] that will bring about the change we are all talking about i say that **ac**taly**p**t [catalyst] is the democratic party and our nomine**d** [nominee] for president we are not strangers to change twenty years ago we changed the whole tone of the nation after wateri**a**te **ca**uses [watergate abuses] we did that years ago we know how to change we have been the instrumental change in the pa**r**t [past] we know what needs to be **f**o**od** [done] and how to do it we know we can impact policies in education human rights civil rights economic and social opportunity and the en**t**ironment [environment] the**r**e [these] are policies which are **hoc**eeded [imbedded] in the soul of our party and i**nc**eeded [imbedded] in our soul they will not disappear easily..."

The results analysis leads to the following conclusions:

- The accuracy reached using n4 model is high enough to make the system practical for use. This is correct especially when the decoded massage is eventually viewed by a person which can guess what are the mistakes and what is the correct decoding using the meaning and the context of the sentence. The 4.5% mistakes can be more problematic when the far side is a machine which responds to the message without human intervention.
- A model with more parameters is, as expected, more accurate than a smaller one.
- The improvement in both model entropy and system accuracy seems to be asymptotic with respect to model size. That is, the impact of enlarging a large model is minor. The small improvement from h3 to h4 might indicate that the size of 40000 parameters reached is not much far from the maximum accuracy rate possible with a PHT based model. On the other hand however, the examination of the example shows that some of the mistakes

involves a choice of non English words. This indicates that the correct words are probably not included in the PHT as complete strings. Hence, further enlarging of the model might add those words to the PHT and remove the resulting mistakes.

- The variable memory models are demonstrated to be considerably more accurate than the fixed memory models with the same size.
- The variable memory model h2 that was built using the hybrid algorithm is slightly more accurate than n2, the model that was created using the greedy algorithm. h3 and n3 gave the same results for the application although h3 is a bit superior in its information content with respect to n3.

6.2 Handwriting Recognition

Several models of portable pen-supported computers with or without keyboard were introduced lately and can be purchased at a cost only slightly higher than other portables. Although many of them still seem to suffer from disappointing handwriting recognition capabilities they do hint to the power and convenience of pen based interface combined with good handwriting recognition systems. The natural 'feel' of such interface is a promising factor in the forecast of this kind of computers. Many groups around the world in both academic institutes and commercial companies are working on developing good handwriting recognition systems.

The intended host for a handwriting recognition system is, in many cases, a small portable computer. This fact presents a set of limitations and demands from the system. These limitations might be quite extreme since the handwriting recognition is only an 'assistant task' to the main running application (word processor, database application etc.). It may consume only small portion of the computer resources which are not wide to begin with. Our system has been built to give answer to those demands:

- Memory limitation - the implementation is small and needs small runtime space (see section 5.5). It gives good results (see below) with a limited set of parameters.
- Model size flexibility - The model size is flexible and it can be set to a desired size without any change in the algorithm that uses it. This makes the same algorithm portable between platforms without losing the benefits of a large model on the stronger machine.
- Speed - The algorithm is fast (see section 5.5).
- On-line behavior - There is only small 'wait to result' gap (see section 5.5).

The two last points are especially important since the handwriting recognition should be 'transparent' to the user. An unexpected 'idle' time between the moment a sign is written and the moment it is being resolved and echoed (as a character) to screen is not acceptable by the users who are used to the zero delay of keyboards.

A version of our system using a PHT-based model and the ATR algorithm described in section 5.4 was implemented in the commercial handwriting recognition system of ART ltd. The description of the complete recognition system is out of the scope of this work. However, this system was built up of several layers. Our algorithm constitutes the final layer which chooses one possible interpretation of a written character out of several options, using the language statistical knowledge combined with other information from previous layers. ART's non-cursive handwriting recognition system is a user dependent trainable system. The tested version of the system achieved a performance of 95.2% correctly recognized characters **without** any statistical language information. This number was raised by our system to 97.0%. This is a reduction of 37.5% of the original 4.8% error rate.

The test was performed on a set of about 24000 signs which were written by 24 different users on a portable computer screen using an electronic pen. The recognition accuracy was measured for a full character set which includes upper and lower case letters, digit, and all other standard keyboard characters. Some of the texts in the test set were not well formed English text. This includes, for example,

sequences like "aBcDeF...", "2 4 6 8 0" or "! @ # $ % ^", mathematics expressions such as "2+3=5" etc.

Following is a table reporting the overall accuracy of the recognition system using different PHT models in the 'linguistic' filter layer (the ATR algorithm). Although the differences seem minor when looking at the percentage of correct characters they are very meaningful when observing the error rate. The low error rate is very important for a usable portable system with no attached keyboard. The error reduction column presents the reduction of error (in percentage) achieved from the original 4.8% error.

model	nodes	entries	entropy	error reduction	accuracy (%)
no linguistic model	0	0	4.906	0.0%	95.2%
f1	1	30	4.213	12.5%	95.8%
f2	31	760	3.456	20.8%	96.2%
f3	776	7594	2.841	25.0%	96.4%
n1	41	753	3.342	18.8%	96.1%
n2	570	7600	2.665	27.1%	96.5%
n3	1792	19744	2.337	35.4%	96.9%
n4	**4300**	**40010**	**2.117**	**37.5%**	**97.0%**
h1	42	753	3.341	18.8%	96.1%
h2	568	7606	2.664	27.1%	96.5%
h3	1802	19743	2.334	35.4%	96.9%

Table 6.2: Application results of PHT-based ATR algorithm in the handwriting recognition system of ART Ltd. The table shows the overall recognition accuracy of the system and the error reduction achieved by the ATR algorithm using different model sizes and types. The n4 base model reduces error by 37.5%. Error reduction is measured in all cases compared to the 4.8% error of ART system without the "linguistic layer".

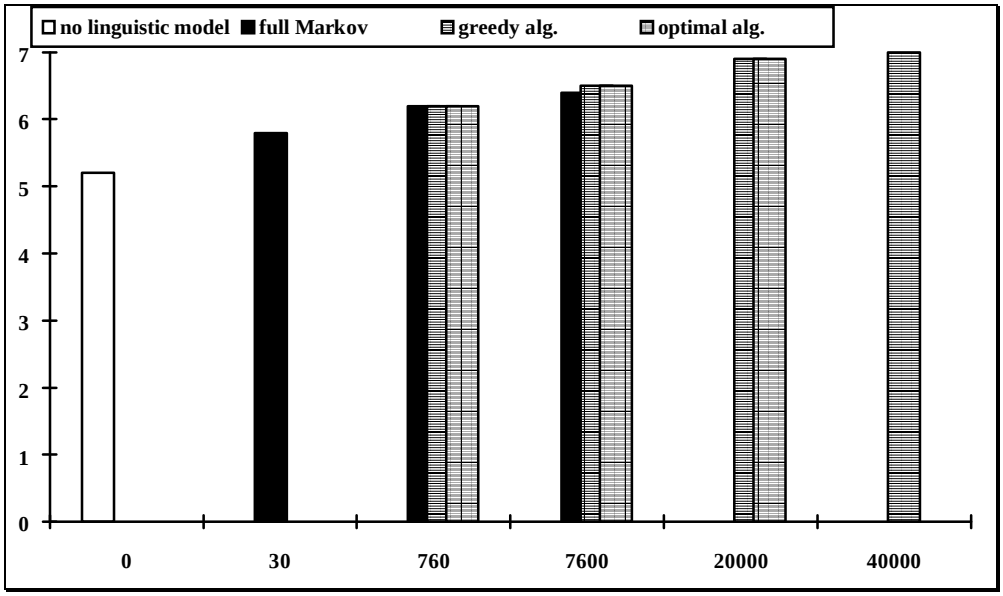


Figure 6.2: Overall character recognition accuracy of the system using different classes of models and different sizes. The y axis is the accuracy of the system in percentage starting from 90% and ending in 97%. The x axis is the size of the model. Attached bars represent models with same sizes but different types.

Some conclusions rise from the analysis of the results:

- Linguistic-based filtering is helpful for handwriting recognition application. This is not trivial since each layer which is added to a system inserts some noise to the overall performance. Since we started up with a relatively good system with no linguistic-based layer (95.2% accuracy) we were not granted to get any improvement in recognition. We want to emphasize that some linguistic knowledge is already used in the non filtered system. This means that the filter and the layers beneath are partly using the same information. Achieving error reduction by the filter is even harder in such situation.

  The linguistic-based layer does not have the **potential** to correct **all** the 4.8% recognition errors. This is since is serves as a filter which chooses one character of each input vector. These vectors are the output of the layers beneath, and are not guarantied to contain the correct character. We found out that the correct character appears in 99.6% of the vectors.

- The overall performance of the system with the filter (97%) is quite satisfactory. A short message, such as an announcement, a memo or an information query can be entered using an electronic pen with none or nearly no mistakes in recognition. If the system supports good user interface for easy mistake fixing then our experience indicates it can be practically used for many application without any keyboard usage at all.
- The larger models gave better performance, as expected, than the smaller ones. The improvement in performance with respect to size seem to be, as in the case of the phone keypad encoding system, asymptotic. The variable memory models are again more accurate than the fixed length memory models with the same size.
- The variable memory models that were built using the hybrid algorithm showed no superiority in performance over the greedy model with the same size. This can be caused by inaccuracy in the results measurements or simply by the mild difference in information content between the two types of models.

Our system proved to be successful in the commercial handwriting recognition system. However, we believe it can obtain even higher accuracy if combined in the recognition process not just as a last filter but rather as one source of information for a general decision procedure which combines some sources for an overall recognition of a text.

Bibliography

[Abe 92] N. Abe and M. Warmuth.
"On the computational complexity of approximating distributions by probabilistic automata",
Machine learning, volume 9, pages 205-260, 1992.

[Bhal 83] Lalit R. Bhal, Frederick Jelinek and Robert L. Mercer.
"A maximum likelihood approach to continuous speech recognition",
IEEE Trans. Pattern Analysis and Machine Intelligence, volume PAMI-5, number 2, pages 179-190, March 1983.

[Brill 92] Eric Brill.
"A simple rule-based part of speech tagger",
Second Conference on Applied Natural Language Processing, Trento, Italy, 1992.

[Chen 92] M. Chen, A. Kundu and J. Zhou.
"Off-line handwritten word recognition (HWR) using a single contextual hidden Markov model" *Proceedings of the IEEE 1992 Computer Society Conference on Computer Vision and Pattern Recognition*, Pages 669-672, Champaign, IL, USA, 1992.

[Church 88] Kenneth W. Church.
"A stochastic part program and noun phrase parser for unrestricted text",
Second Conference on Applied Natural Language Processing, Austin, Texas, 1988.

[Cover 91] Thomas M. Cover and Joy A. Thomas.
"Elements of information theory",
Wiley series in telecommunications, New York, 1991.

[Fano 63] R. M. Fano.
"A Heuristic Introduction to Probabilistic Decoding", *IEEE Trans. Information Theory,* volume IT-9, page 64, 1963.

[Goldschmidt 95] O. Goldschmidt and Dorit S. Hochbaum.
"K-edge Subgraph Problems",
Technical report, IEOR Department, University of California, Berkeley, 1995.

[Höffgen 93] K.-U. Höffgen.
"Learning and robust learning of product distributions",
Proceedings of the Sixth Annual Workshop on Computational Learning Theory, pages 97-106, 1993.

[Huang 93] Xuedong Huang, Fileno Alleva, Hsiao-wuen Hon, Mei-Yuh Hwang, Kai-Fu Lee and Ronald Rosenfeld.
"The SPHINX-II speech recognition system: an overview.",
Computer, Speech and Language, volume 2, pages 137-148, 1993.

[Hull 82] J. J. Hull and S. N. Srihari.
"Experiments in Text Recognition with Binary n-Gram and Viterbi Algorithm",
IEEE Trans. Pattern Analysis and Machine Intelligence,
volume PAMI-4, number 5, pages 520-530, September 1982.

[Jelinek 69] Frederick Jelinek.
"Fast sequential decoding algorithm using a stack",
IBM J. Res. Develop. , volume 13, pages 675-685, 1969.

[Jelinek 77] Frederick Jelinek, Robert L. Mercer, Lalit R. Bhal, and James K. Baker.
"Perplexity - a measure of difficulty of speech recognition tasks",
94th meeting of the Acoustic Society of America, Miami Beach, Florida, USA, December 1977.

[Jelinek 91] Frederick Jelinek and John Laferty.
"Computation of the probability of initial substrings generation by stochastic context-free grammars",
Computational Linguistics, volume 16 part 3, pages: 315-323, September 1991.

[Krogh 93] A. Krogh, S. I. Mian and D. Haussler.
"A hidden Markov model that finds genes in E. Coli DNA",
Technical Report UCSC-CRL-93-16, University of California at Santa-Cruz, 1993.

[Kupiec 92] J. M. Kupiec.
"Robust part-of-speech tagging using a hidden Markov model",
Computer Speech And Language, volume 6, pages 225-242, 1992.

[Lari 91] K. Lari and S. J. Young.
"Applications of stochastic context-free grammars using the Inside-Outside algorithm",
Computer Speech And Language, volume 5, pages 237-257, 1991.

[Nadas 84] Arthur Nadas.
"Estimation of probabilities in the language model of the IBM speech recognition system",
IEEE Trans. on Acoustics, Speech, and Signal processing, volume ASSP-32, number 4, pages 859-861, August 1984.

[Orlowski 89] M. Orlowski and M. Pachter.
"An algorithm for the determination of the longest increasing subsequence in a sequence",
An International Journal: Computers & Mathematics, with Applications, Volume 17, Pages 1073-1075, 1989.

[Rabiner 86] L. R. Rabiner and B. H. Huang.
"An introduction to hidden Markov models",
IEEE ASSP Magazine, pages 4-16, January 1986.

[Rabiner 89] L. R. Rabiner.
"A tutorial on hidden Markov models and selected applications in speech recognition."
Proceedings of the IEEE, 1989.

[Rissanen 83] Jorma Rissanen.
"A universal data compression system",
IEEE Trans. Information Theory, volume IT-29, number 5, pages 656-664, 1983.

[Rissanen 86] Jorma Rissanen.
"Complexity of Strings in the Class of Markov Sources",
IEEE Trans. Information Theory, volume IT-32, number 4, pages 526-532, July 1986.

[Ron 95] Dana Ron, Yoram Singer and Naftali Tishby
"The power of amnesia: learning probabilistic automata with variable memory length",
Machine Learning, volume 1, pages 1-26, 1995.

[Ross 70] S.M. Ross,
"Applied Probability Models with Optimization Applications",
Holden-Day, San Francisco, 1970.

[Schabes 93] Yves Schabes, Michael Roth, and Randy Osborne.
"Parsing the Wall Street Journal with the inside-outside algorithm",
Sixth Conference of the European Chapter of the Association for Computational Linguistics (EACL '93), Utrecht, the Netherlands, April 1993.

[Shannon 48] C. E. Shannon.
"A mathematical theory of communication",
Bell Sys. Tech. Journal, volume 27, pages 379-423, 623-656, 1948.

[Shannon 51] C. E. Shannon.
"Prediction and entropy of printed English",
Bell Sys. Tech. Journal, volume 30, pages 50-64, 1951.

[Skiena 94] Steven S. Skiena and Harald Rau.
"Dialing for document: an experiment in information theory",
Seventh ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '94), Marina Del Rey, California, November 1994.

[Tao 92] Chongguang Tao.
"A Generalization Of Discrete Hidden Markov Model And Of Viterbi Algorithm" ,
Pattern Recognition, Volume 25, number 11, pages 1381-1387, 1992.

[Viterbi 67] A. J. Viterbi.
"Error bounds for convolutional codes and an asymptotically optimal decoding algorithm",
IEEE Trans. Information Theory, volume IT-13, pages 260-269, 1967.

[Weinberger 82] M. J. Weinberger, A. Lempel and J. Ziv.
"A sequential algorithm for universal coding of finite-memory sources",
IEEE Trans. Information Theory, volume IT-38, pages 1002-1014, May 1982.

[Weinberger 95] M. J. Weinberger, J. Rissanen and M. Feder.
"A universal finite memory source",
Technical report, Department of Electronics and Systems Engineering, Tel-Aviv University, Israel, 1995.