A FORMULA TO CALCULATE PRUNING THRESHOLD FOR THE PART-OF-SPEECH TAGGING PROBLEM

Nguyen Chi Hieu

Industrial University of Ho Chi Minh City,
12 Nguyen Van Bao, Ward 4, Go Vap District, Ho Chi Minh City

Email: *nchieu@hui.edu.vn*

Received: 1 May 2016; Accepted for Publication: 15 July 2016

ABSTRACT

The exact tagging of the words in the texts is a very important task in the natural language processing. It can support parsing the text, contribute to the solution of the polysemous word, and help to access a semantic information, etc. One of crucial factors in the POS (Part-of-Speech) tagging approaches based on the statistical method is the processing time. In this paper, we propose an approach to calculate the pruning threshold, which can apply into the Viterbi algorithm of Hidden Markov model for tagging the texts in the natural language processing. Experiment on the 1.000.000 words on the tag of the Wall Street Journal corpus showed that our proposed solution is satisfactory.

Keywords: Hidden Markov model, Part-of-speech tagging, Viterbi algorithm, Beam search.

1. INTRODUCTION

The tagging is defined as an automatic assignment of descriptors (or tags) to input tokens. Part-of-speech (POS) tagging is a selecting process to find the most likely sequence of syntactic categories for words in a sentence. It is a very important problem in natural language processing. Several approaches have been developed [1], which include taggers based on handwritten rules, n-gram automatically derived from tagged text corpora, Hidden Markov models, symbolic language models, machine learning, and hybrid taggers [2].

Among the above approaches, one based on the Hidden Markov model (HMM) can offer prominent results [3]. Especially, when using the Viterbi algorithm, it can achieve an accuracy rate of over 95 percent [4]. However, its complexity is a challenge. For a problem involving T words and K lexical categories, the algorithm which is to find the most likely sequence requires $K^2 * T$ steps for bigram and $K^3 * T$ steps for trigram. When K increases, the original Viterbi algorithm becomes slow.

For this reason, several investigators have developed techniques to try to reduce the generation time and memory consumption in practice. They are based on the heuristic pruning methods. However, the heuristic pruning techniques may miss essential parts of the search tree, caused by wrong decisions while pruning. Therefore, in this paper, we propose a method to calculate the pruning threshold, *and* apply it into the Viterbi algorithm of HMM. Beam search is a heuristic method for the combinatorial optimization problems, which has extensively been studying in artificial intelligence [5]. The method combining the Viterbi algorithm with Beam pruning technique is useful to compress the search space, which reduces the computational complexity [6]. The base difference of our solution from traditional heuristic pruning techniques is that the search pruning formula is proposed instead of using a heuristic.

2. RELATED WORKS

2.1. Hidden Markov model

In 1913, Andrey Markov asked a less controversial question about Pushkin' text: could we use frequency counts from the texts to help compute the probability that the letter in sequence would be the vowels. Afterwards, Hidden Markov models are applied in temporal pattern recognition as speech, handwriting, gesture recognition, part-of-speech tagging, partial discharges and bioinformatics, etc. and named a Markov process [7]. The Hidden Markov model is one of the most important machine learning models in speech and language processing. A Markov chain is a special case of the Markov process when states can be numbered. Figure 1 shows a Markov chain for assigning a probability to the sequence of categories, for which the category consists of ART, N, V, and P. A Markov chain embodies an important assumption about these probabilities. In the first-order Markov chain, the probability of a particular state depends only on the previous state [4].

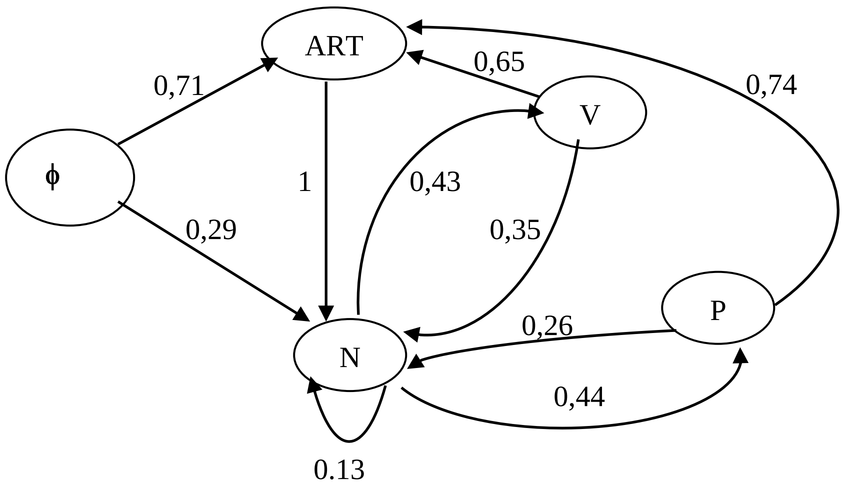

Figure 1. A Markov chain.

The Markov chain is a random process that undergoes transitions from one state to another on a state space. It must possess a property that is usually characterized as memoryless: the probability distribution of the next state depends only on the current state and not on the sequence of events that preceded it [8]. This specific kind of memoryless is called the Markov property. A stochastic process has the Markov property if the conditional probability distribution of future states of the process depends only upon the present state, not on the sequence of events that preceded it. A process with this property is called a Markov process. The Markov property can be defined as follows: suppose $S = \{S_0, S_1 \dots S_n\}$ are states in state-space, $S_i \in W$ $1 \leq t \leq n$, where W is the finite set of possible symbols. We said that S_i has Markov property if the probability: P (S_{t-1} | S_t) = P (S_0 S_1 …S_{t-1} | S_t).

A Markov chain is useful when we need to compute a probability for a sequence of events that we can observe in the real world. In many cases, however, the events we may not be directly observable. For example, part-of-speech tagging, a HMM allows us to talk about both *observed* events (like words that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in probabilistic model [7]. The word *hidden* means unobserved states in the Markov model. For example, the sequence words: *the flies like flowers*, the word *flowers* could be generated from state N with a probability of 0.05, or at state V with a probability of 0.053 [4, page 200]. Like this, a probability of string "*the flies like flowers"*, with a different category is the difference.

2.2. Beam Search

A beam search ranks all the alternatives produced so far by their probabilities, and always just expand the most likely sequences [9]. This continues to be done at each time point until the entire output is covered by an interpretation. This is a simple modification to the basic Viterbi algorithm.

It can be seen that the original Viterbi algorithm was driven by a loop that traserves each position in the input, namely

$$\delta_t(i) = \text{Max}_{j = 1,N} (\delta_{t-1}(j) \times P(C_i \mid C_j)) \times P(W_t \mid C_i), \tag{1}$$

where, $W_t \in W = W_1W_2 \ldots W_T$ is a sequence of words, $C_i \in \hat{C} = C_1C_2 \ldots C_K$ is a sequence of lexical categories, $\delta_t(i)$ is a probability of W_t with a category C_i, Pr (C_i | C_j) is a probability of a category C_i in context of lexical category C_j, Pr (W_t | C_i) a probability of word W_t in context of lexical category C_i.

All *K* states (the i's) were looped and a Max calculation of a formula (1) over all *K* predecessor states (the j's) was done by the process for each input position, i.e., K^2 operations. The unpromising states at every step are pruned out by a beam search, leaving only the best nodes to calculate the next states. By using several different methods, we can decide how many to keep. We, for example, make a decision to keep a fixed number at each step, say *D* nodes. Alternately, we could choose the factor *H* and delete all states that have a probability less than *m*H*, where *m* is the maximum probability for a state at that time. For instance, we would only retain hypotheses that the score is greater than the half of the maximum one at that time if *H = 0.5*.

For more detail, we decide to keep all nodes that $m*K \geq k$ threshold at each step. We create an array called *beam*, the set indices of states above threshold at time *t* will be stored by each element of which. The formula (1) have been presented with

$$\delta_t(i) = \text{Max}_{j \in \text{Beam}(t-1)} (\delta_{t-1}(j) \times P(C_i \mid C_j)) \times P(W_t \mid C_i) \tag{2}$$

Then, by finding the maximum value M_t at time *t*, we can compute *beam(t)* and delete the states below the threshold of M_t*H. The number of operations in the inner loop to be *D*K* has been reduced with this change, where *D* is the average number of states above threshold from M_t to M_t*H. We can make this a manageable number by setting *v*. For instance, with the lexical-generation and bigram probabilities, we take the problem of finding the most likely categories for the sentence *"the flies like flowers"* into consideration. It is showed that there are 4 possible categories and 4^4 = 256 different sequences of length four.

We consider the part of speech tagging example for the sentence "*the flies like flowers"*. In

Table 1, this sentence presented the Viterbi algorithm's result with the Markov hypothesis. As expected, four nodes are enlarged at each stage in which consider four achievable paths, and four stages. It means that 4^3 possibilities are calculated. A transition diagram in Figure 2 represents all 256 sequences.

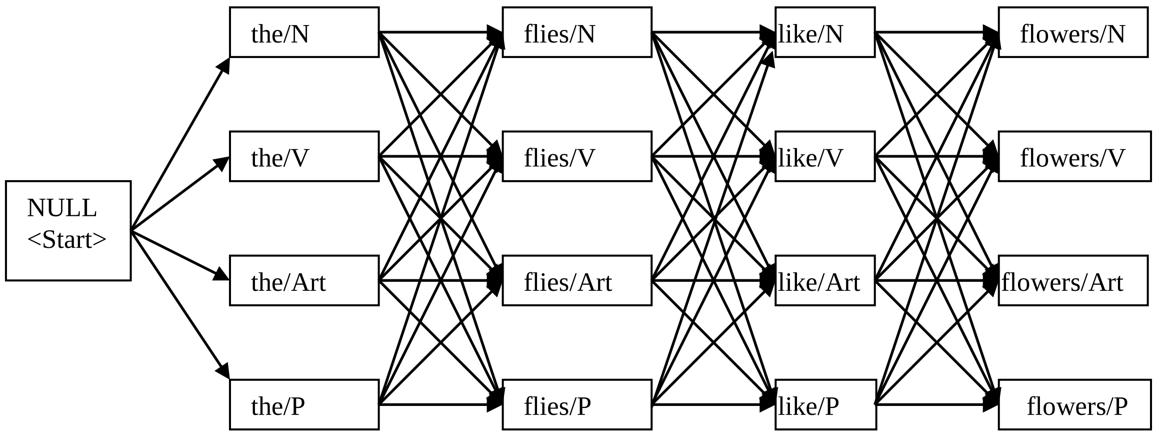

Figure 2. Encoding the 256 possible sequence "*the flies like flowers*".

Table 1. The full Viterbi search for "*the flies like flowers*".

	the	**flies**	**like**	**flowers**
N	$0.29\ e^{-4}$	$\mathbf{0.95\ e^{-2}}$	$0.15\ e^{-4}$	$\mathbf{0.7\ e^{-5}}$
V	$0.1\ e^{-7}$	$0.95\ e^{-5}$	$\mathbf{0.41\ e^{-3}}$	$0.34\ e^{-6}$
Art	**0.38**	$0.38\ e^{-8}$	$0.62\ e^{-9}$	$0.27\ e^{-7}$
P	$0.6\ e^{-8}$	$0.128\ e^{-8}$	$0.28\ e^{-3}$	$0.66\ e^{-9}$

Table 2. The Beam - search Viterbi employing a factor *H = 0.01* for "*the flies like flowers*".

	the	**flies**	**like**	**flowers**
N		$\mathbf{0.95\ e^{-2}}$	$\mathbf{0.15\ e^{-4}}$	$\mathbf{0.7\ e^{-5}}$
V			$\mathbf{0.41\ e^{-3}}$	$\mathbf{0.34\ e^{-6}}$
Art	**0.38**			
P			$\mathbf{0.28\ e^{-3}}$	

Table 2 indicates the beam - search Viterbi just retaining entries that are within the factor of 0.01 for the flies like flowers. In this table, there are three stages, which retain just one entry. During the search, the algorithm considers 5×4 = 20 possibilities [10]. Table 3 presents a fixed beam width of two (i.e., the two best ones at each point are kept) of the beam search Viterbi.

Table 3. The set beam - search Viterbi with *w = 2* for *the flies like flowers.*

	the	**flies**	**like**	**flowers**
N	**$0.29\ e^{-4}$**	**$0.95\ e^{-2}$**		**$0.7\ e^{-5}$**
V		**$0.95\ e^{-5}$**	**$0.41\ e^{-3}$**	**$0.34\ e^{-6}$**
Art	**0.38**			
P			**$0.28\ e^{-3}$**	

Here there are two entries for every stage, thus the algorithm calculates 8*4 = 32 possibilities.

3. PROPOSED APPROACH

Pruning in the beam search can be classified into two broad categories. The first approach, called complete anytime beam search was introduced by Zhang [11]. This algorithm calls a beam search, and if the beam search fails to find a solution, it tries again with a larger beam. Eventually this algorithm will find a solution because at some point the beam will become so large that no important thing is pruned. This approach has the serious drawback that the amount of space required to search to a given depth increases as time passes. An alternative approach is to expand pruned nodes without restarting the search. This is the approach taken in Limited discrepancy beam search [12], and beam-stack search [13]. Both beam-stack search and Limited discrepancy beam search reconsider pruned nodes without restarting the search, but they do so in a very systematic manner that does not consider heuristic information. Both algorithms approaches allow a complete exploration of the search space, but this completeness comes at a cost.

Using a too narrow or tight beam (too low beam width D, a factor h) in Viterbi beam search can prune the best path, and may return errors [10]. But using a too large beam is unnecessary for computation in searching unlikely paths. That is disadvantage of search methods using heuristic. In the formula (2), *pruning beam search techniques* only focus on $\delta_{t-1}(j)$, and determine pruning threshold by experimental. This paper proposes a formula to calculate pruning threshold $(\delta_{t-1}(j)*P(C_i|C_j))$ automatically, and apply it into the Viterbi algorithm of HMM.

3.1. The Viterbi Algorithm

With the exhausted search, we have to search the maximum N^T possibles for a sentence of T words and N tags for each word. Luckily, it is unnecessary to enumerate N^T paths because we can construct the dynamic programming algorithms for the above problem. One of these algorithms is the Viterbi Algorithm of HMM.

To take the searching the above HMM on the input of the sentence *the flies like flowers* into consideration. We have many steps to deal with:

- To compute a maximum probability rather than a minimum distance

- To find routes at each point in time. With an HMM, it is necessary to generalize the algorithm to solve probability distributions over states and to keep a separate record for each state at each time point (record of state C_i at time i will have a different record of state C_i at time i-1).

- To look for the best path whatever state it ends in.

The details of the algorithm are shown in Figure 3. This algorithm operates by computing the values of these two arrays that are SeqScore (i,t) and BACKPTR(i,t). The algorithm will search the probability of the best sequence leading to each possible category at each position in the search space. We use an *N*x*T* array for this space, where *N* is the number of lexical categories ($C_1…L_N$) and *T* is the number of words in the sentence ($w_1..w_T$). The SeqScore(i, t) array records the maximum probability of a word W_t in category C_i. Another *N*x*T* array is the BACKPTR(i,t) that will indicate for each category in each position what the preceding category is in the best sequence at position i-1.

Given word sequence W_1, ..., W_T, lexical categories C_1, ..., C_N , output probabilities Pr (W_i | C_i) and bigram probabilities Pr (C_i | C_j), find the most likely sequence of $C_1,…,C_T$ for the word sequence $W_1,…, W_T$.

```
Initialization Step:
for i = 1 to N do        /*N  is the number of lexical categories; <Start>: NULL/ø */
        SeqScore(i,1) = Pr(C_1 | <Start>)* Pr(W_1 | C_i)
            BACKPTR(i,1) = 0;
Iteration Step:
    for t = 2 to T do             /* T is the number of the word in a sentence */
    for i = 1 to N do
SeqScore (i,t) = Max (SeqScore (j,t -1)* Pr (C_i | C_j))* Pr (W_t | C_i), với j = 1,..N
BACKPTR(i,t) =  index of j that gave the max above
Sequence Identification Step:
    C(T) = i is Max of SeqScore(i,t)
    for i = T-1 to 1 do
C(i) = BACKPTR(C(i+1),i+1)
```

Figure 3. The Viterbi algorithm.

3.2. Integrating pruning calculation threshold into the Viterbi algorithm

The time (or the number of steps) which takes to complete a program depends on some factors such as the code of compiler, state and compute code, data, big O, etc.

3.2.1. Dataset

We use a part of the Penn Treebank corpus (about 1,000,000 words annotated with POS). The Penn treebank POS tag set has 36 POS tags plus 12 others for punctuations and special symbols. 36.65 % of words have more than one lexical category (average is 2.44 lexical categories per a word). And 63.35 % of words have one lexical category. The average value is 1.47 lexical categories per a word for total 1,000,000 words [14]. This is the problem of spare data. Therefore, its probability in this category could be calculated as 0, so the probability of the overall sentence containing the word would be 0 too. For this reason, they have other calculated techniques to address the problem of estimating the above low probabilities. The technique to solve the zero probability problem is that we might add a small amount, say 0.5 to every count before the estimating probabilities Pr(Wt | Ci), and assign for Pr(Ci | Cj) = 10^{-6} with probabilities of Pr(Ci | Cj) = 0. This guarantees that there are not the retaining zero probabilities. They called this estimation technique is the expected likelihood estimator (ELE) [4].

3.2.2. Algorithm analysis

Analysing the base Viterbi algorithm, we have found that, one step t (t = 2,...,T), Viterbi algorithm have to calculate a probability of a word to take out Max a formula (3).

SeqScore(i,t) = Max(SeqScore(j,t -1) Pr(C_i | C_j))* Pr(W_t | C_i), with j = 1..K* (3)

Notation: P = *Max(SeqScore(j,t -1)* Pr(C_i | C_j)), with j = 1..K*

Result of the P formula will be depended on two parts: SeqScore(j,t -1) and Pr(C_i | C_j), which means that we need find two formulas *SeqScore(j,t -1) and Pr(*C_i | C_j*) such that the product SeqScore(j,t -1)* Pr(*C_i | C_j*) is maximal.*

The Breadth–First–Search algorithm [15] with a suitable *heuristic* will make a result search fastest for the SeqScore(j,t -1). However, Pr(C_i | C_j) may be not large enough to create a maximal P. A solution to make the maximal P is presented in section 3.3.

3.2.3 Threshold factor

Although we do not know the *SeqScore(j,t -1)* Pr(*C_i | C_j*)* result when it is maximal, oneself on the data characteristic and the past processing, we can know when to stop process calculate P. The parameter to control this process is called the Threshold factor. With this assumption, a calculation step, t is always made a result like a formula (4).

SeqScore(j,t -1) >= SeqScore(j+1,t -1). (by sorting) (4)

Setting up Pr_{max}(C_i | C_j) is the maximal value in the Pr(C_i | C_j) values, *with j,i = 1,...,N.*

We make an examination of the formula P = *Max (SeqScore(j,t -1)* Pr(*C_i | C_j*)), with j = 1,...,N. We realize that we can stop the calculation of P, if the SeqScore(j,t -1)*Pr(*C_i |C_j*) >= SeqScore(j+1,t -1)**Pr_{max}(C_i| C_j*) in the j-th step. Due to, SeqScore(r,t -1) < SeqScore(j+1,t -1),* where r = j+2…N (by the assumption in a formula (4)).

To reduce the number of the multiplification operations in the algorithm, we change an inequality:

SeqScore(j,t -1)* Pr(C_i | C_j) >= SeqScore(j+1,t -1)* Pr_{max}(C_i | C_j) (5)

By SeqScore(j,t -1)* Pr(C_i | C_j) / Pr_{max}(C_i | C_j) >= SeqScore(j+1,t -1) (6)

Set up *dBeam* = Pr(C_i | C_j) / Pr_{max}(C_i | C_j), (7)

3.2.4. Proposed algorithm – The pruning Viterbi algorithm

We can calculate *dBeam*, before the program is run, and we call the *dBeam,* Threshold factor. In addition, we add a variable to rank for the *Pr (C_i | C_j)* that is called the *iBack*. The details of the proposed algorithm are shown in Figure 4.

The difference between the base Viterbi algorithm and the pruning Viterbi algorithm is in the Iteration Step. In addition, we create an established Heap algorithm to get the biggest element in *SeqScore(t,N)* array in such a way that it is fastest. We use the statistical method to compare the time it takes to complete the program between the base Viterbi algorithm and the pruning Viterbi algorithm.

```
❖ Step 1: Preparing data
  – To sort the probabilities $Pr(W_t \mid C_i)$ in the decrease order, this task helps to make *heap*
    faster, due to we do not need to change the position of the elements (following the strategy to
    optimize partiality [2]).
  – To calculate rank and threshold of components in tab $Pr(C_i \mid C_j)$.
❖ Step 2: Using pruning technology in the Iteration Step of Viterbi algorithm
(i)    for t = 2 to T do
(ii)     for i = 1 to K do
           begin
    (1)      SeqScore(t –1,K-i); //convert SeqScore(t –1,K-i) into heap// O(logK)
    (2)        PMax = SeqScore(i,t -1)* $Pr(C_i \mid C_1)$;
    (3)        max=1;
    (4)        if ((rank of $Pr(C_i \mid C_1)$>1) and
                   (SeqScore (0,t -1)* $Pr(C_i \mid C_0)$.dBeam < SeqScore(1,t-1)))
             then
(iii)           for j=2 to K do
                begin
    (5)           if (SeqScore(j,t -1)* $Pr(C_i \mid C_j)$.dBeam >= SeqScore (j+1,t -1))
                   then                // exit for with j argument;
    (6)          A = SeqScore(j,t -1)* $Pr(C_i \mid C_j)$;
    (7)      if (A>PMax) then
                 begin
    (8)            PMax=A;
    (9)            max = j;
               end;
             end;
    (10)     SeqScore(i,t) = PMax* $Pr(W_t \mid C_i)$;
    (11)     BackPtr(i,t) = max;
           end;
```

Figure 4. The pruning Viterbi algorithm

4. EXPERIMENTAL RESULT

We used a part of the Penn Treebank corpus for training and testing [16]. We divided the corpus into two parts: the training and test set. We chose randomly the 3000 sentences in the WSJ part of the Penn Treebank corpus that sentences have different length. In addition, attaching the variable counts to determine the *If* statements and assignment statements in the pruning Viterbi algorithm. By assuming that an *If* statements and an assignment statement have the same run time, as shown in Table 4.

Table 4. A summary of the statements count in the test.

N_o	Words in sentence	Quantity sentences	T*K*logK (E) (Heap)	Results				Rate D/(A+E)
				A	B	C	D	
1	3	250	189750	285241	1140456	398843	2071326	4.36
2	5	250	316250	676012	2159305	1153451	4168350	4.20

3	8	250	506000	1063105	3534163	1993250	7263122	4.63
4	10	250	632500	2743247	5042016	5077515	10468054	3.10
5	16	250	1012000	2704276	7120150	5124199	15570318	4.19
6	20	350	1771007	4545821	12835032	80905650	27589426	4.37
7	24	250	1518000	4287091	11584866	7862927	24988512	4.30
8	32	250	2024000	5352826	14153251	13044250	32290542	4.39
9	38	250	2403500	11679096	19414311	21777512	37841566	2.68
10	40	250	2530000	8336102	19731752	16142521	40692570	3.75
11	53	250	3352250	9784750	25789183	18259751	57221084	4.36
12	62	150	2352975	7537491	17847771	14254069	38719195	3.90
Total:		3000	18608232	58995058			298884065	3.85

where: A: There are iBack and dBeam; B: There is not iBack, and there is dBeam; C: There is iBack, and there is not dBeam; D: There are not iBack and dBeam; E: Times counting to create Heap O (*T*K*logK*).

For a problem involving *T* words and *K* lexical categories, the base Viterbi algorithm is guaranteed to find the most likely sequence using $T*K^2$ steps. In other words, Big O of the base Viterbi algorithm is the $O(T*K^2)$. In the Penn Treebank, $K = 46$;

Notation:

$$Tg_{origin} = O(T*K^2) = D$$

$$Tg_{pruning} = A + E$$

$$Tg_{origin} / Tg_{pruning} = D/(A + E) = 3.85 \qquad (8)$$

where: Tg_{origin}: is the run time of the base Viterbi algorithm; $Tg_{pruning}$: is the run time of the pruning Viterbi algorithm.

By counting the number of commands to perform two algorithms Viterbi algorithm the base (Tg_{origin}) and the Viterbi algorithm pruning ($Tg_{pruning}$) and compare the statements executed by the formula (8), the results showed that the pruning algorithm Viterbi algorithm implementation of commands less than the base algorithm Viterbi algorithm. But to get the full survey on the effectiveness and speed of calculation, the article needs further experimentation on the other Tree banks. On the other hand, the contribution of this paper is made by formula *beam* threshold. This threshold is calculated based on the probability of both components $\delta_{t-1}(j) * P(C_i \mid C_j)$ instead of having to select the threshold from empirical through $\delta_{t-1}(j)$.

5. CONCLUSION

In this paper, we propose an approach to calculate the pruning threshold, and apply it into the original Viterbi algorithm. The basic difference of our solution from traditional heuristic pruning techniques is that the search pruning method is proposed instead of using a heuristic. Although the result of the proposed solution is rather satisfactory on the Wall Street Journal, this proposal should carry out the experimental study on the other Treebanks.

REFERENCES

1. Ruslan Mitkov - Computational Linguistics, The Oxford University Press, First Published, 2003, 219-220.
2. Padro L. - A hybrid environment for syntax-semantic tagging. Ph.D thesis, Departament de Llenguatges i Sistemes Informatics, Universitat Politecnica de Catalunya, Barcelona, 1998.
3. Yuan L.C. - Improved hidden Markov model for speech recognition and POS tagging, Journal of Central South University **19** (2012) pp. 511-516.
4. James Allen - Natural Language Understanding, The Benjamin/Cummings Publishing Company, Inc., 1995, pp. 144-155.
5. Pinedo M. - Scheduling: Theory, algorithms, and systems, Prentice-Hall, Second Addition, 2002.
6. Liu W. and Han W. - Improved Viterbi algorithm in continuous speech recognition, International Conference on Computer Application and System Modeling (ICCASM) **7** (2010) 201-207.
7. Daniel Jurafsky and James H. Martin - Speech and Language processing: an introduction to natural language processing, computational linguistics, and speech recognition – 2nd ed. Prentice Hall, 2013, 139-149.
8. Philipp Koehn, Pharaoh - A beam search decoder for phrase-based statistical machine translation models, Machine translation: From real users to research, 2004, pp. 115–124.
9. Christoph Tillmann, Hermann Ney - Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation, Journal Computational Linguistics **29** (1) (2003) 97-133
10. Jame Allen, Seaching HMM model, http://www.cs.rochester.edu/u/james/CSC248, 2003
11. Zhang W. - Complete anytime beam search, Proceedings of AAAI-98, 1998, pp. 425–430.
12. Furcy D. and Koenig S. - Limited discrepancy beam search, Proceedings of the International Joint Conference on Artificial Intelligence, 2005, pp. 125–131.
13. Zhou R. and Hansen E. A. - Beam - stack search: Integrating backtracking with beam search, Proceedings of ICAPS-05, 2005, 90-98.
14. Ferran Pla, Antonio Molina and Natividad Prieto - Tagging and chunking with bigrams. In: Proceedings of the 18th conference on Computational linguistics **2**. Association for Computational Linguistics, 2000, 614-620.
15. Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest - Introduction to Algorithms, The MIT Press, 1990, 594-602.
16. The Penn TreeBank Project. [Online].

    Available: ftp://ftp.cis.upenn.edu/pub/treebank/doc/update.cd2