A* HIERARCHICAL TREE PRUNING PARSING IN VIETNAMESE TEXT-TO-SPEECH

Le Quang Thang[1], *Tran Do Dat*[1,2], *Nguyen Thi Thu Trang*[2]
1. International Research Institute MICA CNRS UMI 2954 – Hanoi University of Science and Technology, INPG
2. School of Information and Communication Technology – Hanoi University of Science and Technology

ABSTRACT

This paper describes how to improve the speed of Vietnamese Speech Synthesizer by improving the speed of the Vietnamese parsing system. First, a briefly chapter about parsing system using A* algorithm will be introduced. After that, we will discuss about the hierarchical tree algorithm for improving the speed of parsing system. With this algorithm, the speed problem of parsing system can be solved.

Categories and Subject Descriptors
Knowledge-based and information systems
General Terms
Algorithm, languages
Keywords
A*, parsing, hierarchical tree

1. INTRODUCTION

There are many parsers which have been researched in Vietnam [5], [6], [7]. In our knowledge, there is no research about the A*search algorithm for parsing. The other best-first-search algorithm like beam search, Dijkstra… were implemented and got the nice result [6], [7]. However, these algorithms are still lame and got some troubles about speed and accuracy. A* algorithm is always considered higher than those algorithms not only in parsing, but also in the searching. So, A* algorithm based parser is a good choice to research and development.

A* algorithm is a well-known algorithm in the best-first-search branch, which is one of the best searching algorithm in the world [11]. Unlike the other best-first-search, the perfect of A* algorithm makes it become the algorithm model which researched and improved by many scientists. A* algorithm for parsing has been proposed by Dan Klein and Christopher D. Manning (2003) and got some good result about improving the speed of parsing system[2].

In this paper, we present about two issues. First, we describes the A* algorithm which is adapted to parsing technique in Vietnamese Speech Synthesizer. And second, the HTA (hierarchical tree algorithm) model is proposed to improving the speed of A* algorithm in parsing system.

2. A* ALGORITHM FOR PARSING

A* algorithm operates on basically parsing items called node. A node includes three attributes: *tag, start and end*. Tag indicates the current node POS tag (known as syntactic tag), and *(start, end)* denote a start-end position of the string which the *node* generates in the sentence. The parser maintains *two* data structures: a chart or table (note as CHART), which records edges for which (best) parses have already been found, and an agenda of newly-formed edges to be processed (note as AGENDA).

2.1. The A* parsing process

First, the input sentence is processed by a tokenizer and a tagger to provide a set of node, it is AGENDA.

Second, the maximum candidate node is popped out from AGENDA for processing. If it isn't contained in CHART, it will combine with these nodes in the CHART. These combinations generate more nodes to append to AGENDA. And the last, the candidate will be added to the CHART.

The loop of second step will be repeated until one of those conditions is reached: (1) the AGENDA is empty or (2) the *node* S (1, n) is found in CHART (with n is the number of tokens in the input sentence).

2.2. A* estimates for parsing

It's plain to see the most important thing in the A* parsing algorithm would be the estimates for the maximum candidate.

The detail of the A* estimates for parsing is described in [3].

3. HIERARCHICAL TREE ALGORITHM

3.1. The problem

In the second step of A* algorithm, the candidate will combine with the element *node* in the CHART. But the problem here is how to combine and how is the speed of combination?. A lame combination algorithm will form a lame parsing.

The classic method for this combination is using the virtual node processing. It means that the parser combines the candidate one-on-one with each *node* in the CHART. There are two situations that happened:

The founded grammar rule is a Chomsky-form, means that the rule have no more than two elements on the right part. These *nodes* have just been combined in ordinary way.

The founded grammar rule is not a Chomsky-form; the grammar rule has more than two elements on the right part. When this case happened, the parser uses a virtual node with the wait parameter which denotes the lack part to complete the rule. It means when A and B are combined together using a rule like "E → A B C D", they will form the *node* (E, wait = "CD"). Later, if the virtual node (E, wait="CD") meets C *node* with relevant position, two *nodes* will combine together and form the *node* (E, wait="D").

So, when the parsing process ends, if the *node* (S, 1, n, wait="") is founded in CHART, the parsing process would be failed and vice versa.

The virtual node can solve the problem of combination (how to combine) but not the speed of combination. Cause of the complication of the grammar rule set (approximately over 900 rules!!), the combination using virtual node will generate a very large number of redundant *node.* Specified as table 1, the combination of two elements has formed so many new elements, and usually only few of them are used.

Table 1 – all the *nodes* was formed when combined N(2,7) and V(7,8)

NP(2,8, wait="")	*NP(2,8, wait=", A")*
NP(2,8, wait="AP")	*NP(2,8, wait="AP NP")*
NP(2,8, wait="AP PP")	*NP(2,8, wait="MP")*
NP(2,8, wait="N")	*NP(2,8, wait="NP")*
NP(2,8, wait="NP PP")	*NP(2,8, wait="NP VP")*
NP(2,8, wait="P")	*NP(2,8, wait="PP")*
NP(2,8, wait="PP PP")	*NP(2,8, wait="VP")*

3.2. Fundamental hierarchical tree algorithm (HTA)

3.2.1. Basic ideal.

Instead of using virtual node method, hierarchical tree algorithm processes all the position-combinable chain of the candidate with CHART. All the position-continuous chain including a candidate *node* and the *nodes* in CHART will be checked whether it is the right part of any grammar rule or not? The right-checked-chain will form new *nodes* using the relevant grammar rule. Unlike virtual node method, the hierarchical tree algorithm does not form the redundant *nodes* and decrease the number of loop steps of A* algorithm.

For instance, if the candidate has the start-end position as X(7-10) and the CHART has the content in Table 2.

Table 2 – All the start-end pos of CHART *nodes*

X_1 (1-8)	X_2 (6-16)	X_3 (15-35)	X_4 (5-20)	X_5 (2-7)	X_6 (10-11)
X_7 (8-27)	X_8 (2-21)	X_9 (9-11)	X_{10} (2-13)	X_{11} (6-14)	X_{12} (15-26)
X_{13} (14-23)	X_{14} (5-18)	X_{15} (1-7)	X_{16} (9-16)	X_{17} (12-17)	X_{18} (7-18)
X_{19} (6-25)	X_{20} (13-26)	X_{21} (11-16)	X_{22} (9-24)	X_{23} (11-20)	X_{24} (8-18)
X_{25} (7-16)	X_{26} (14-16)	X_{27} (4-6)	X_{28} (13-21)	X_{29} (4-8)	X_{30} (11-13)

With the input (candidate and CHART *nodes* position), the hierarchical tree algorithm will process and provide the output chain to check that is presented in Table 3:

Table 3 – all the combinable chain of the candidate and CHART *nodes.*

1	Position	*node*
2	[2-7] **[7-10]**	X_5**X**
3	[1-7] **[7-10]**	X_{15}**X**
4	[1-7] **[7-10]** [10-11] [11-13]	X_{15} **X** X_6 X_{30}
5	[2-7]**[7-10]**[10-11] [11-13] [13-26]	X_5 **X** X_6 X_{30} X_{20}
6	**[7-10]** [10-11] [11-20]	**X** X_6 X_{23}
7	**[7-10]** [10-11] [11-13]	**X** X_6 X_{30}
8	**[7-10]** [10-11] [11-13] [13-26]	**X** X_6 X_{30} X_{20}
9	**[7-10]** [10-11] [11-13] [13-21]	**X** X_6 X_{30} X_{28}
10	[2-7] **[7-10]** [10-11]	X_5 **X** X_6
11	[2-7] **[7-10]** [10-11] [11-16]	X_5 **X** X_6 X_{21}
12	[2-7] **[7-10]** [10-11] [11-20]	X_5 **X** X_6 X_{23}
13	[2-7] **[7-10]** [10-11] [11-13]	X_5 **X** X_6 X_{30}
14	**[7-10]** [10-11] [11-16]	**X** X_6 X_{21}
15	[1-7] **[7-10]** [10-11]	X_{15} **X** X_6
16	[2-7] **[7-10]** [10-11] [11-13] [13-21]	X_5 **X** X_6 X_{30} X_{28}
17	[1-7] **[7-10]** [10-11] [11-16]	X_{15} **X** X_6 X_{21}
18	[1-7] **[7-10]** [10-11] [11-20]	X_{15} **X** X_6 X_{23}
19	**[7-10]** [10-11]	**X** X_6
20	[1-7] **[7-10]** [10-11] [11-13] [13-26]	X_{15} **X** X_6 X_{30} X_{20}
21	[1-7] **[7-10]** [10-11] [11-13] [13-21]	X_{15} **X** X_6 X_{30} X_{28}

Thus, assuming that there is a rule as A → X_5 **X** X_6 X_{30} X_{28} relevant to the chain 16, the *node* A(2,21) will form with the relevant position.

3.2.2. The proposed HTA model

HTA includes two steps: classification and generation of combination chains.

Classification [(1)]: the period when the parser classifies the *nodes* of CHART into the difference block. This classification is a preparation for chain generating period.

Generation of combination chains (GOCC) [(2)]: the parser generates all the combinable chain and processes each of them for the next stage.

3.2.2.1. Classification step

The HTA classification is based on pigeon hole sort algorithm ideal. The HTA system creates the holes for adding pigeon. But the holes in HTA is used for GOCC[(2)] instead of sorting.

The holes in HTA are divided into two type: the left holes and the right holes (Figure 1). Assuming that X is a candidate *node.*

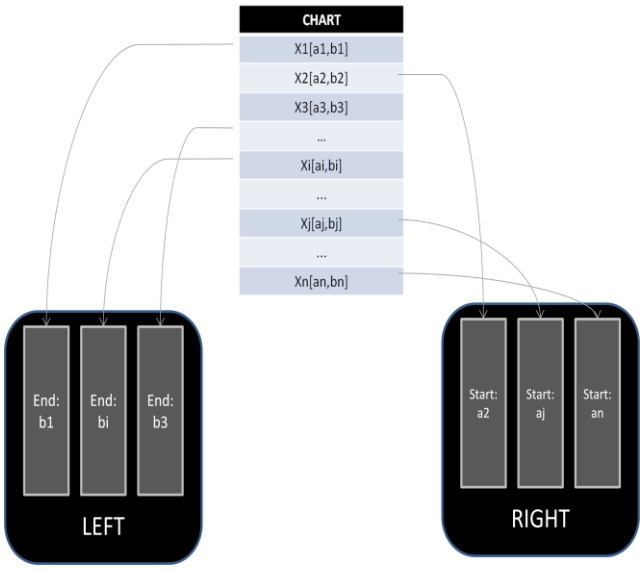

Figure 1 – Classification model in hierarchical tree.

- Left holes: This is a set of *nodes* that have their *end* position <= *start* position of X. All the *nodes* with the same *end* position equal *e* will be added in the one block labeled as *e*. And all of the blocks lay in the bigger "left holes".
- Right holes: This is a set of *nodes* that have their *start* position >= *end* position of X. All the *nodes* with the same *start* position equals *s* will be added in the one block labeled as *s*. And all of the blocks lay in the bigger "right holes".

3.2.2.2. Generate the combination chains

With the input as the classified CHART, the parsing system will begin generating the chain. It includes three parts: "generate left chain", "generate right chain" and "generate chain". (show in Figure 2)

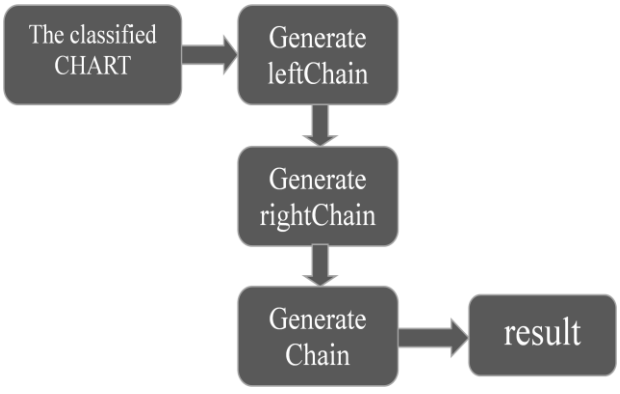

Figure 2 – the HTA chain generator.

Generate left chain: this module will generate all the combinable chain ends with candidate X, described as:

- Access to the left block labeled as "start position of X", called the left combination nodes of X, note as SX block.
- Process the entire *node* in SX recursively. With each node, generate a chain relevant to that *node*.

On the other hand, "generate left chain" process resemble to the tree processing with root as X. When the *node* Y is processed, the generated chain is the path from Y to X on the tree. (Figure 3)

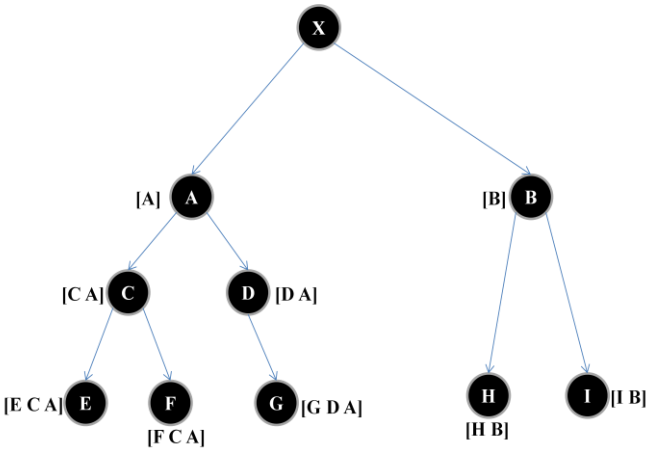

Figure 3 – The instance example for generate left chain.

Generate right chain: same method as the "generate left chain", only differ from the dimension.

- Access to the right block labeled as "end position of X", called right combination nodes of X, note as EX block.
- Process all of the *nodes* in EX recursively. With each node, generate a chain relevant to that *node*.

Resemble to the left chain, "generate right chain" process resemble to the tree processing with root as X, too. But when the *node* Y is processed, the generated chain is the path from X to Y on the tree.

Generate chain: so now we got the left chain and right chain of the candidate. It's a perfect preparation for the party. The connection of three factors "left chain", "right chain" and X will form the real combinable chain of X.

First, we will generate the chain ends with X using the left chain.

For (**left** *in* **leftChain***)*

Generate **chaini** = [left X]*;*

*Chain. Add (***chaini***);*

End for;

Second, we will generate the chain starts with X using the right chain.

*For (***right** *in* **rightChain***)*
Generate **chainj** =[X right]*;*
*Chain. Add (***chainj***);*
End for;

And the last, we will generate the chain with X in the mid using both left and right chain.

*For (***right** *in* **rightChain***)*
*For (***left** *in* **leftChain***)*
Generate **chaink** = [left X right]*;*
*Chain. Add (***chaink***);*
End for;
End for;

After three stages like above, we got the real combinable chain to process and perform the A* parsing algorithm using HTA.

3.3. Pruning for HTA

As mentioned above, HTA is proposed to increase the speed, to decrease the step of the A* algorithm. However, with all thing was described, HTA is not optimal cause of the processing time for each step. A* using HTA has less step than the A* - virtual node but the processing time for the step of HTA could be so long because the parser must process all the combinable chain of the candidate with CHART.

In fact, there are approximately 8% of the generated chains that could be combined by the grammar rule. About this stuff, virtual node algorithm and HTA has the same feature: redundancy! The A* virtual node got the redundancy of the loop step, but the A* using HTA got the redundancy of the processing chains. Because of this, HTA is not only slower than virtual in some case, but also very slow when the number of CHART *nodes* up to 500 elements. In addition, the system could be out of memory in case the number of chains is up to billions!

To solve this problem, the paper writer proposed the pruning method for HTA, note as "HTA pruning". Instead of processing all the generated chains, the parser will prune the chains that don't make sense; it means they're not relevant to any rule. This algorithm is not only increase the speed of parser but also optimize it.

From now on, we will use some abbreviation symbol; it's convenient for audience to allow.

R_{chain} – A set of grammar rules, each rule contains "chain" in the right part.

F_{chain} – A set of grammar rules, each rule have the right part starts with "chain".

HTA only uses the *node* position to generate chain, but not the *node* tag. For instance, if the *node* has the tag as "PP", so what kind of its tag will be? It's plain to see that HTA make a big mistake when it doesn't calculate this case.

Through the analysis above, the tag of *node* is also very important in the chain generating process. So the pruner of HTA will use the information about the tag of node in the grammar rule to optimize the algorithm.

The pruner of hierarchical tree includes two parts:

- Statistic training: this is a very important work. Because of this stage, the parser will decide whether to prune the branch of processing tree
- Pruning: start processing HTA; use the pruner which was trained from data to prune the wrong branch.

3.3.1. Statistic training period

The stuff required for each trainer is a training data. And in this case, the training data is a syntactic grammar rule set.

Specifically, with each POS tag in the grammar rule, the system will create a corresponding data tree. The data tree of each tag T will store the information about all tags that can be stood on the left or right of tag T.

3.3.1.1. The left-side data tree

The left data tree of tag T is the data structure which store the information about all tags are on the left side of T.

Specified as the picture below:

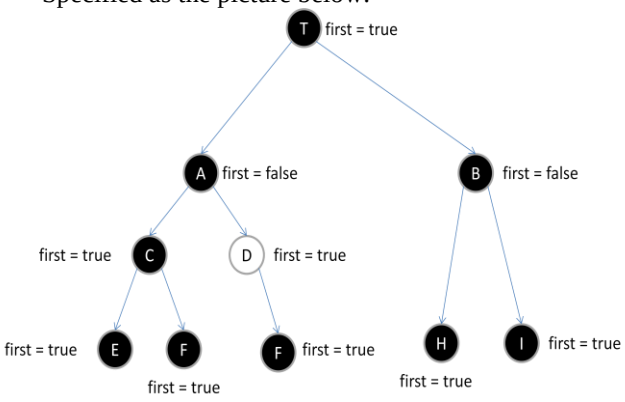

Figure 4 – the left data tree of T.

There are three steps to create this data structure:

- *First*, the trainer will process $R_{[T]}$. The two sub nodes A and B of T are created as the tags that adjacent to the left of T in the $R_{[T]}$ rules right side.
- *Second*, C and D are created as the sub node of A which are the tags that adjacent to the left of A in the set $R_{leftTag(A)}$. Similarly, H and I are the tags that adjacent to the left of B in the set $R_{leftTag(B)}$ rules right side.

  leftTag(node) = path from node to root
- *And the last*, process recursively with the entire sub node C, D, H, I and their sub nodes until it ends.

And one more thing, each node of the tree has a very important thing, the first parameter. It's a boolean type. If the first of tag C equals true, it message that the grammar rule has a existence of at least rule which have leftTag(C) heading in its right side. The tag node has the first equals true which called the *first node*.

So, the purpose of this work is to control the information about the tag of left chain that can be created by T tag.

3.3.1.2. The right-side data tree

With each first node in the left data tree will have the right-side data tree. The right data tree will have the information about the tags which stand on the right of that first node leftTag.

For example, the D tag node in the left tree data shown as picture 4. Its right data tree will help us controlling information about all the grammar rules that D leftTag heads in their right side. leftTag(D) = [D A T], the rule set which have each rule starts with [D A T] or leftTag(D), we note it as $F_{[D\ A\ T]}$.

The picture below will show us a perspective visual about the right-side data tree.

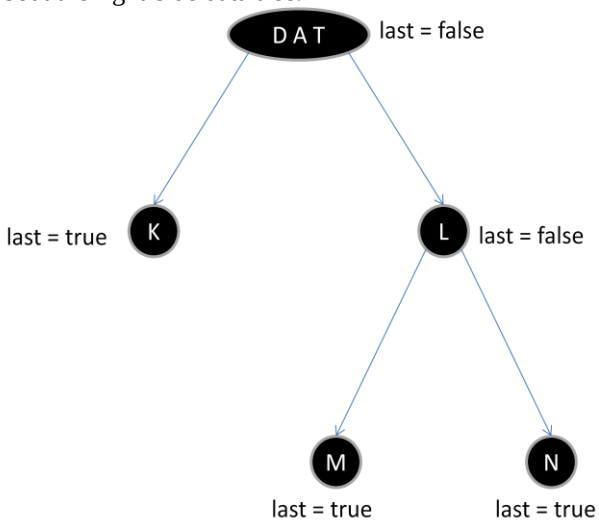

Figure 5 – the right-side data tree of D tag node.

The root node has two children K and L which are the tags adjacent to the right of T in $F_{[D\ A\ T]}$. Resemble to the left data tree, M and N are the tags adjacent to the right of L in $F_{rightTag(L)}$.

rightTag(node) = leftTag(root) + path from root to that node.

rightTag(L) = [D A T L]

The right data tree also has the critical parameter: last. It's a Boolean type, too. If the last of node equals true, it message that rightTag(node) must be right side of at least one rule in the grammar rule set. For instance, M has the true last, it indicate that rightTag(M) = [D A T L M] is a right side of one or some grammar rule.

So, after creating the left and right data trees, we have the information about all the tags in grammar rule and we can control their relation to prune the redundancy branch in HTA.

3.3.2. Pruning period

The input is still the classified CHART; HTA will perform normal, adding the support of HTA pruning.

The pruner will not only prun all the redundancy branch, but also indicate when we got the exactly we got the right side of any rule due to first and last parameter.

For instance, as the picture shown below which express the left chain generation of HTA. In the process, the C node was prunned because it didn't appear in children set of A in the left data tree of X, so it wan's any tag that adjacent to the left of A in the grammar rule, and it was prunned. This can be useful because most of tree node in the chain generating process will be prunned for the redundancy, and increase the speed of HTA to the best of it.

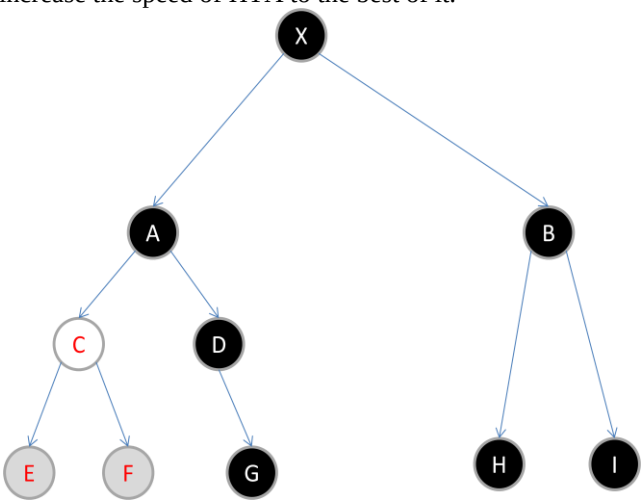

Figure 6 – the HTA pruning process.

4. EXPERIMENT AND RESULT

This section presents the preparation and the result of experiment to demonstrate the A* parsing algorithm performance. These two subsections below will summarize our activities and results for experiment.

4.1. Preparation for experiment

As described, the most important thing of parsing system in our target is to increasing the speed up to maximum as it can, and the second one is accuracy [5], [6]. We also add one more goal for the system: analyzable. Because the Vietnamese grammar is too difficult to analyze, so that the percent of analyzed sentences is not high. Due to three goals like that, we conduct an experiment with two tests for estimating the quality of system:

- First: the test with 630 Vietnamese sentences in mica database - vnSpeechCorpus to test the speed of A* parser. This dataset includes long and difficult sentences which is a challenge to any parsing system.
- Second: The accuracy testing corpus which we choose is extracted from the training set of system – VietTreeBank. VietTreeBank has been built by VLSP Groupment, has included 20.000 sentences of Vietnamese which has been parsed by hand. Within a range of this paper, the quantity we used is about 200 sentences.

And one last thing, in order to presenting the audience the performance, we also make a comparison between A* parsing algorithm with very well-known search parsing algorithm: CYK-Beam search.

4.2. Results of experiment

The first test:

Algorithm	Processing time	Number of parsed sentence
A*	15 minutes	92%
CYK-Beam search	45 minutes	75%

The second test:

Algorithm	Accuracy	Number of parsed sentence
A*	70%	92%
CYK-Beam search	50%	75%

As you can see, A* parsing is better than CYK-beam search in all field. With high speed, significantly number of parsed sentence and an acceptable accuracy, the A* parser is really a good parser.

These test based on the A* algorithm without the support of hierarchical tree algorithm. Due to rush time, we haven't finished it yet, but to proposed its ideal. This stuff show that if the A* hierarchical tree is completed, it will bring us a much better result than originally A* algorithm. The time could be decreased down to a few minutes with the same testing set.

5. CONCLUSIONS

The paper presented all ideal about our proposed algorithm: hierarchical tree. A hierarchical tree algorithm was implementing in java and makes some test. Its speed is so amazing, but it still independent to the A*. The hierarchical tree algorithm will make a new speed generation of the parser. With the most important thing that a real-time Vietnam speech synthesizer system needs is a speed, A* hierarchical tree algorithm is really an excellent idea.

6. REFERENCE

[1]. **Fei Xia**, "*Inside-Outside algorithm*", LING 572.

[2]. **Christopher D.Manning and Hinrich Schutze. "***Probabilistic Grammars*", Chapter11, 1999.

[3]. **Dan Klein and Christopher D. Manning**. 2003. "*A* parsing: Fast exact Viterbi parse selection. In Proceedings of the Human Language Technology Conference and the North American Association for Computational Linguistics*"(HLT-NAACL).

[4]. **Dan Klein and Christopher D. Manning**. 2002. *"A* parsing: Fast exact Viterbi parse selection"*. Technical Report dbpubs/2002-16, Stanford University, Stanford, CA.

[5]. **Hoàng Anh Việt**, "*Phân tích cú pháp tiếng Việt sử dụng mô hình xác suất PCFG*", đồ án tốt nghiệp đại học năm 2006.

[6]. **Phạm Thị Nhung**, "*Phân tích cú pháp tiếng Việt sử dụng beam search*", đồ án tốt nghiệp đại học năm 2009.

[7]. **Đỗ Bá Lâm, Lê Thanh Hương**, "*Implementing a Vietnamese syntactic parser using HPSG*", Khoa Công nghệ thông tin, trường Đại học Bách khoa Hà Nội.

[8]. **Diệp Quang Ban, Hoàng Văn Thung,** "*Ngữ pháp tiếng Việt*", tập 1,2, Nhà xuất bản giáo dục, 1991-1992.

[9]. **Trung tâm khoa học xã hội và nhân văn Quốc Gia. "***Ngữ pháp tiếng Việt***".** Nhà xuất bản Khoa học Xã hội – 2000.

[10]. **Nguyễn Phương Thái, Vũ Xuân Lương, Nguyễn Thị Minh Huyền.** *"Xây dựng treebank tiếng Việt".*

[11]. *http://en.wikipedia.org/wiki/A*_search_algorithm*