A* HIERARCHICAL TREE PRUNING PARSING IN VIETNAMESE LANGUAGE

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 our research on development of a Vietnamese text parser using *A* hierarchical tree pruning parsing* algorithm. The algorithm is composed of two algorithms: the A* parsing algorithm and a hierarchical tree pruning algorithm. The A* algorithm was integrated in our Vietnamese parsing system. The parsing system has a much higher speed and accuracy (>20%) in comparison with a system using traditional algorithms, such as Beam-search. In addition, we proposed a new hierarchical tree pruning algorithm, based on this algorithm, the speed of parsing system can be improved.

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

1. INTRODUCTION

There are some 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. The A* algorithm is always considered better than these algorithms not only in parsing, but also in the searching. So, an A* algorithm based parser is a good choice to research and development.

The A* algorithm is a well-known algorithm in the best-first-search branch, which is one of the best searching algorithms in the world [8]. The A* algorithm for parsing was proposed by Dan Klein and Christopher D. Manning (2003) and got a good result in improving the speed of parsing system [3], [4].

Two main issues are presented in this paper. Firstly, 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 is not 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 S(1, n) *node* is found in CHART (with n is the number of tokens in the input sentence).

2.2. A* estimates for parsing

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 it has *less-than or equal to* two elements on the right part. These *nodes* have just been combined in ordinary way.

The founded grammar rule is not a Chomsky-form; it 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 that when A and B nodes are combined together using a rule like "E → A B C D" (for example), they will form a virtual *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 idea.

Instead of using virtual node method, hierarchical tree algorithm uses *combinable chains* in each loop step of the A* parsing algorithm. A *combinable chain* is a sequence of nodes which meets the following condition: the end-position of a node must be equaled to the start-position of next node For example, a simple combinable chain: (NP[1, 3] PP[3, 5] VP [5,8]).In the HTA, all the combinable chains including a candidate *node* and the *nodes* in CHART will be checked whether it is the right part of any rule in the set of syntactic grammar rules or not? The satisfied-chains will form new *nodes* using the relevant grammar rule. Unlike virtual node method, the hierarchical tree algorithm does not form the redundant virtual *nodes* and it decreases the number of loop steps of A* algorithm.

For instance, 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)

From this input data (candidate and CHART *nodes* position), the hierarchical tree algorithm will process and generate the combinable chains which are presented in Table 3:

Table 3 – all the combinable chains of the candidate with 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 16th chain in the table 3, the *node* A(2,21) will be created and will be added to AGENDA.

3.2.2. The proposed HTA model

The HTA model includes two phases: classificator and generator of combinable chains.

Classificator [(1)]: the parser classifies the *nodes* of CHART into the difference blocks.

Generator of combinable chains (GOCC) [(2)]: the parser generates all the combinable chains and uses them to create a new node which is added into AGENDA.

3.2.2.1. Classificator phase

The HTA classificator is based on pigeon hole sort algorithm idea. 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 the HTA are divided into two types: the *left hole*s and the *right hole*s (Figure 1). Let assuming that X is a candidate *node*.

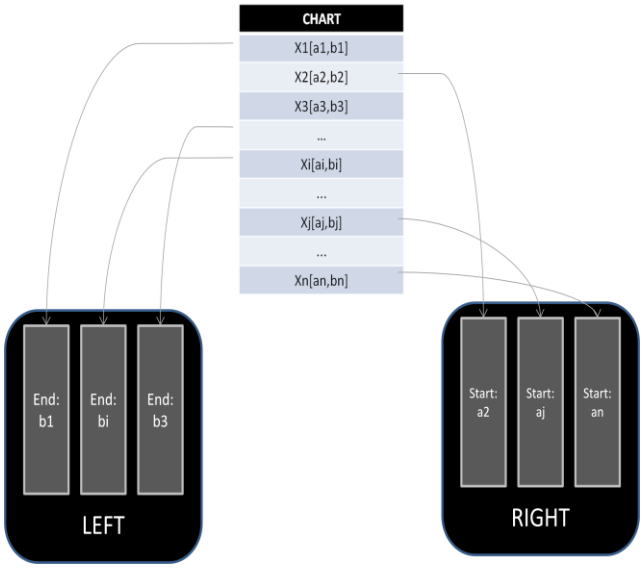

Figure 1 – Classification model in hierarchical tree.

- *Left hole*: This is a set of *nodes* that have their *end* position <= *start* position of X. All the *nodes* which have the same *end* position equaling to *e* (for instant) will be added in a block labeled as *e*. And a set of all these blocks is labeled as a *left hole*.
- *Right hole*: This is a set of *nodes* that have their *start* position >= *end* position of X. All the *nodes* that have the same *start* position equaling to *s* (for instant) will be added in a block labeled as *s*. And a set of these blocks is called as a *right hole*.

3.2.2.2. Generator the combinable chains

With the input as the classified CHART, the parsing system begins generating the combinable chains. It includes three phases: "generating *left chains*", "generating *right chains*" and "generating combinable chains". (Figure 2)

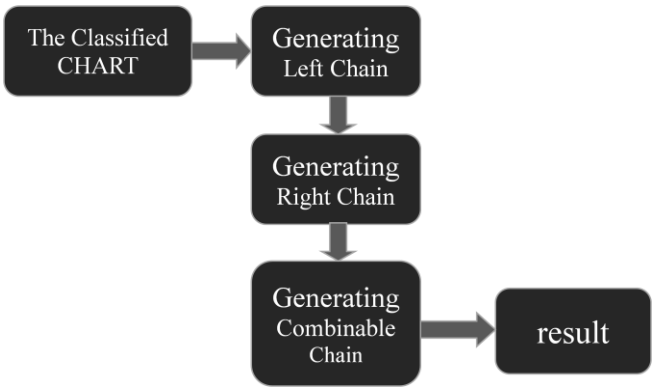

Figure 2 – The Generator the combinable chains of HTA.

1. Generating left chains: this module generates all the combinable chains which end with candidate X, it is called as the *left chains*. We imply that S(E) is the block in a ***left hole*** which is labeled as a start position of node E. This phase can be described as:

- Parsing system accesses the S(X) block and process all the nodes in this block.
- This progress is done recursively for all the *nodes* in the S(X) until satisfying a specific condition. This condition is: "*there is not any block in left hole to access*". This progress resembles to the *tree traversal* at X root. When the *node* Y is processed, *a g*enerated combinable chain is the path from Y to *X on* the tree. For example, for the E node, a relevant generated combinable chain is the path from E to X which is named as **[E C A]**. (Figure 3)

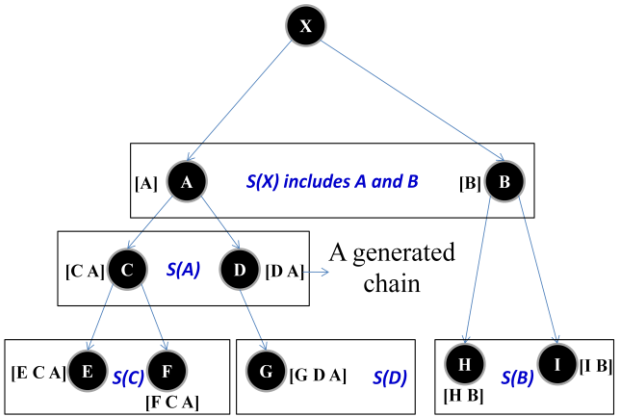

Figure 3 – The instance example for generating *left chains*.

2. Generating right chains: the same progress as the "generating *left chains*" is realized.

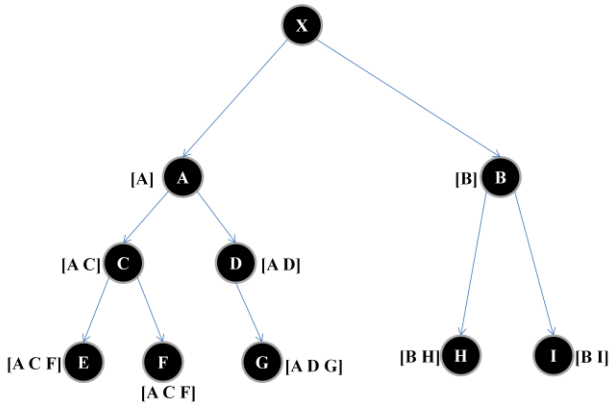

Figure 4 – The instance example for generating *right chain*s.

We imply that E(S) is the block in a ***right hole*** which is labeled as a start position of node S. This phase can be described as:

- Parsing system accesses the E(X) block and process all the nodes in this block.
- Resemble to the *left chain*s, "generate *right chain*s" process resemble to the tree traversal with root as X, too. But when the *node* Y is processed, a generated combinable chain is the path from X to Y on the tree (Figure 4).

3. Generating combinable chain*:* from two first phases we got the *left chain* and *right chain* of the candidate. The connection of three factors "*left chain*", "*right chain*" and X will form the real combinable chains of X.

First, we 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 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 generate the *combinable chain* with X in the middle 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 phases, we got a set of combinable chains to process and to perform the A* parsing algorithm using HTA.

3.3. Pruning for HTA

As mentioned above, the HTA is proposed to increase the speed, to decrease the number of loop steps of the A* algorithm. However, the HTA is still not optimal because of the processing time for each loop step. A number of loop steps of A* using HTA is less than that of the A* using virtual node but the processing time for each loop step of the HTA could be so long because the parser must process all the combinable chains of the candidate with CHART.

In fact, from our experiment on testing performance of HTA, we found that there are approximately 8% of the combinable chains that could be used. Therefore, the virtual node algorithm and the HTA have the same problem: redundancy. The A* virtual node got the redundancy of the number of loop steps, but the A* using HTA got the redundancy of the number of combinable chains in each loop step. Because of this redundancy, the HTA is not only slower than virtual node algorithm 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 if number of combinable chains is up to billions!

To solve this problem, we proposed a pruning method for HTA, noted as "HTA pruning". Instead of processing all the combinable chains, the parser will prune the combinable chains that do not create any new node; it means they're not relevant to any syntactic grammar rule. This algorithm is not only increasing the speed of parser but also optimizing it.

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

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

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

A node in A* parsing algorithm has two informations: position and tag. HTA only uses the *node* position to generate chain, but not the *node* tag.

For instance, if we have two nodes: NP(1,7) and PP(1,7). On the HTA point of view, they are just the same, even their tag is difference. So, a proposed pruning method show you how to use the information of *tag* to reduce the processing time of HTA.

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

The pruner of hierarchical tree includes two phases:

- Statistic training phase: using the set of syntactic grammar rule to train and create a pruner. With

the pruner, the parser will decide whether to prune the branch of processing tree

- Pruning phase: during processing HTA; use the pruner which was trained from data to prune the wrong branch.

3.3.1. Statistic training phase

The training data of the HTA pruner is a set of syntactic grammar rule.

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 and 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 stores the information about all tags being on the left side of T.

Specified as the figure below (Figure 5):

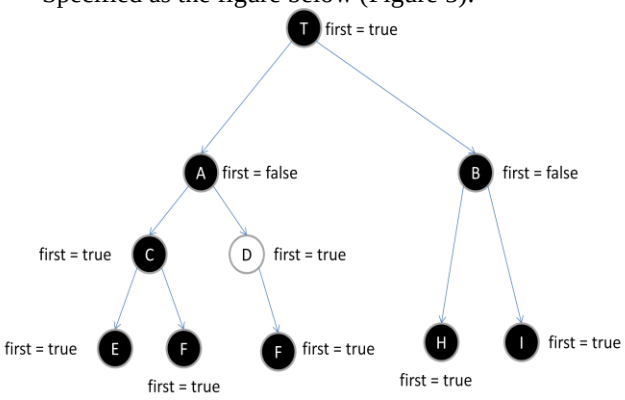

Figure 5 – the left data tree of T.

There are three steps to create this data structure:

- *The first step*, the pruning trainer processes $R_{[T]}$. The two sub nodes A and B of T are created as the tags that are adjacent to the left of T in the $R_{[T]}$ rules right side.
- *The second step*, the C and D tags are created as the children of A. They are the tags that are adjacent to the left of A in the set $R_{left\text{-}tag(A)}$. Similarly, the H and I tags are the tags that are adjacent to the left of B in the set $R_{left\text{-}tag(B)}$ rules *right-side*. And left-tag is a function:

  left-tag(tag) = path from tag to root
- *And the final step, the pruning trainer* processes recursively with the entire children: C, D, H, I and their children until it ends.

In addition, each node of the tree has a FIRST parameter. It's a Boolean type. If FIRST of "C" tag equals true, it indicates that the grammar rule has an existence of at least rule which have *left-tag*(C) heading in its right side. The tag node has FIRST equaling to 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-side data tree will have the right-side data tree. The right-side data tree will have the information about the tags which stand on the right of that *first-node left-tag*.

For example, the D tag in the left-side tree data shown as Figure 5. Its right-side data tree will help us controlling information about all the grammar rules that *left-tag*(D) heading in their right side. *left-tag*(D) = [D A T], all the sets of rule which have each rule starts with [D A T] or *left-tag*(D), we note it as $F_{[D\ A\ T]}$.

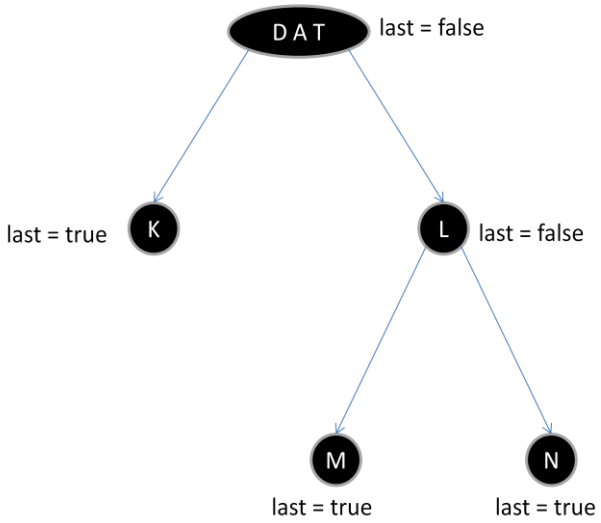

Figure 6 – a right-side data tree of D tag node.

Figure 6 illustrates a right-side data tree of D tag node which is mentioned as above. The root tag has two children K and L tags which are the tags adjacent to the right of T in $F_{[D\ A\ T]}$. Resemble to the left data tree, M and N tags are the tags adjacent to the right of L in $F_{right\text{-}tag(L)}$.

right-tag(node) = left-tag(root) + path from root to that node.

right-tag(L) = [D A T L]

Each right-side data tree has a LAST parameter. It's a Boolean type. If LAST of node equals to TRUE, it indicates that *right-tag* (tag) must be *right-side* of at least one rule in the set of syntactic grammar rules. For instance, M has the true LAST, it indicates that *right-tag* (M) = [D A T L M] is a *right-side* of one or some grammar rules.

So, after creating the left and right data trees, we have the information about all the tags which are in the set of syntactic grammar rules and we can control their relation to prune the redundancy branches in the HTA.

3.3.2. Pruning phase

The input is the classified CHART; the HTA will perform with the support of the HTA pruner.

The pruner not only prunes all the redundancy branches, but also indicates "When does the current chain equal to *right-side* of a specific syntactic grammar rule". The HTA will use these results to create new nodes immediately.For instance, the Figure 7 presents the *left chain* generator of HTA. In this phase, the C node will be prunned because its tag does not appear in child nodes of "A" tag in the *left data tree* of the X tag. Thus, the C tag is not a tag that is adjacent to the left of A in the set of syntactic grammar rules.

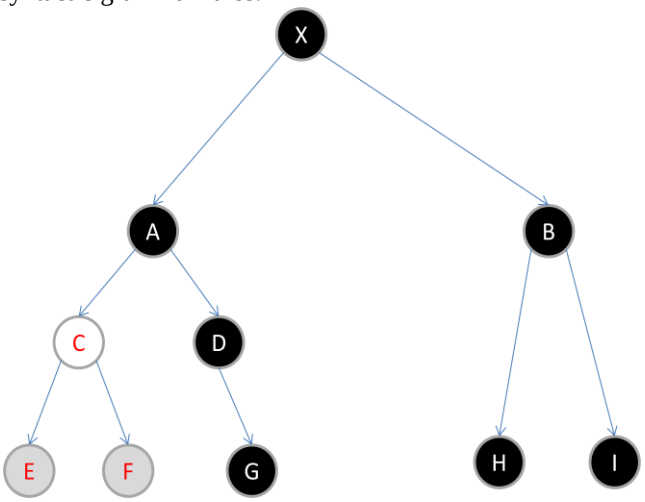

Figure 7 – the HTA pruning process.

4. EXPERIMENT AND RESULT

This section presents the preparation and the result of experiment to demonstrate the A* parsing algorithm performance.

4.1. Preparation for experiment

As described, the most important criteria of parsing system in our target are to increasing the speed, and the second one is accuracy [5], [6]. We also add one more criteria for the system: analyzable. Based on three criteria, we implemented an experiment with two tests for estimating the quality of system:

- The first test is realized to test the speed of A* parser. The test used 630 Vietnamese sentences which is extracted from the text corpus of VNSpeechCorpus [9].
- The second test is implemented to evaluate the accuracy of parsing system. The test corpus is extracted from Viet Treebank database [2]. VietTreeBank has been built by VLSP Group, it includes 2.000 Vietnamese sentences which were parsed manually. In this paper, 200 sentences were used for the test.

We made a comparison between A* parsing algorithm with a very well-known search parsing algorithm: CYK-Beam search.

4.2. Results of experiment

Table 4: the processing time and the analyzable rate

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

Table 5: the accuracy and the analyzable rate

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

The tables 4 and 5 present the results of the first and the second tests consequently. We can see that, A* parsing is better than CYK-beam search in all evaluation criteria. With high speed, significantly number of parsed sentence and an acceptable accuracy, a parser using the A*algorithm is really a good parser for Vietnamese language.

In these tests, the parser has not been yet integrated a proposed hierarchical tree algorithm. We will carry out further experiments to have better assessment results.

5. CONCLUSIONS

The paper presented our study on the development of a Vietnamese parsing system which will be used in Vietnamese TTS. The obtained results showed that the A* algorithm could be integrated for parsing Vietnamese sentences and the A* based parser has a good performance on processing speed, accuracy and analyzable rates. In this study, we proposed a hierarchical tree algorithm for A* based parser in order to improve its parsing speed. This algorithm is integrated into our system and its results will be presented in detail in next studies.

6. REFERENCE

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

[2]. **Website:** *http://vlsp.vietlp.org:8080/demo/?page=resources*

[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]. **Hoang Anh Viet**, "*Vietnamese parsing technique using PCFG*", graduation thesis, HUST, 2006.
[6]. **Pham Thi Nhung**, "*Vietnamese parsing technique using beam-search algorithm*", graduation thesis, HUST, 2009.
[7]. **Lam Do B., Huong Le T**. 2008. Implementing A Vietnamese Syntactic Parser Using HPSG. In Proceedings of the International Conference on Asian Language Processing (IALP), Nov. 12-14, 2008, Chiang Mai, Thailand.
[8]. **Website,** *http://en.wikipedia.org/wiki/A*_search_algorithm*.
[9]. **TRAN D.D.**, **CASTELLI E., TRINH V. L. & LE V.B.**. (2004) *Building a large Vietnamese Speech Database.* Revue vietnamienne « Science et Technologie » (ISBN 0868-3980). Vol. 46/47, February 2004, pp 13-17