**A pruning hierarchical tree method (*PHM*) using in A\* algorithm in Vietnamese parsing technique**

Le Quang Thang

Speech Communication Department

MICA Institute – Ha Noi University of Technology

Ha Noi, Vietnam

E-mail: [quang-thang.le@mica.edu.vn](mailto:quang-thang.le@mica.edu.vn); [lelightwin@gmail.com](mailto:lelightwin@gmail.com)

Tran Do Dat

Speech Communication Department

MICA Institute – Ha Noi University of Technology

Ha Noi, Vietnam

E-mail: [Do-Dat.Tran@mica.edu.vn](mailto:Do-Dat.Tran@mica.edu.vn)

*Abstract* — this paper presents our research on pruning hierarchical tree method in A\* (A-star) in Vietnamese parsing technique in order to improve the speed of Vietnamese parsing system. Based on the virtual node method proposed in [ref], we will describe our replace method: pruning hierarchical tree method. Unlike the virtual node method, PHM process only significant candidates and do not generate new redundancy candidates for each step of A\* algorithm. With this method, the speed of parsing system could be improved so much.

Keywords – A\*, parsing technique, PHM, algorithm, Vietnamese

# Introduction

PCFG and LPCFG in parsing technique are very well-known models. In the parsing system using these models, the final result is determined based on the score of candidates. The candidate with highest score will be the predict outcome. With these models, especially LPCFG, the accuracy of parsing system is relatively high. However, when dealing with wide-coverage grammars and very long sentence, the parsing process is very complicated and the cost for processing time is too expensive. To solve this problem, many parsing algorithm has been researched to reduce the work. Some of them like Beam Search, Greedy Algorithm, and Dijkstra Algorithm. But all of algorithms above also have its problem. Beam search use the beam to remove the underrated candidates, so it is not guaranteed to find the best result. The Greedy Algorithm only follows the best path in each step, so it got a very fast parsing time but it cannot be guaranteed to find the best result, too. The Dijkstra algorithm will find the best result, but its speed, in many cases, is too slow. A\* parsing algorithm which proposed by Dan Klein and Christopher D.Manning could correct those two problems: best result and speed. A\* algorithm is considered as one of the best searching algorithm in the world. It uses a heuristic f(x) to determine the best candidate for each step of parsing process:

f(x) = g(x) + h(x)

In which:

g(x) - the path-cost function, which is the cost from the starting *node* to the current *node*.

h(x) - an admissible "heuristic estimate" of the distance to the goal.

And the most important figure is h(x), it will determine how fast the parsing process leads to the target.

There are many parsing algorithm which have been researched and developed in Vietnamese like Beam search, Greedy algorithm and Machine learning... But in our knowledge, there is no research about A\* parsing algorithm. So, A\* algorithm for parsing is a good choice to research.

In this paper, we present you two main major parts. The first major heading presents about A\* parsing algorithm idea and how to estimate the h(x) parameter. The second major heading which is a mainly focus of our research, presents you about the pruning hierarchical tree method, denoted as PHM. This method is a replacement for the classical virtual node method in order to reducing the estimating cost of parsing process. Therefore, the speed of A\* parsing algorithm could be improved.

# A\* ALGORITHM FOR PARSING

## Basic concept

A\* algorithm operates on basically items called “*node*”. A *node* includes three attributes: *name, start,* and *end.* *Name* attribute indicates the name of *node* (also known as lexical tag or POS [ref]). And the attribute couple (*start, end*) is the start and end position of the text which is covered by *node* in the sentence. Its format is *name* [*start, end*].

Based on this basically items, the parser maintains two data structures: a chart (note as CHART) which records *nodes* for which (best) parses have already been found, and an agenda of newly-formed *nodes* needs to be processed (note as AGENDA).

## A\* parsing process

First, the input string is tokenized into *n* words a1…an. And then, these words are POS tagged to produce an array of *node* called AGENDA:

*{<Xi [i, i+1], wi>,}*

While AGENDA is not empty and CHART does not contain S [1, n+1] (S is a POS of sentence) do

Remove a *node* (<Y,i,j>,w) with highest

w + h(<Y,i,j>)

Second, the maximum candidate *node* is popped out from AGENDA. 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).

# pruning hierarchical tree method

## The context for proposition

In the second step of A\* algorithm, the candidate *node* will combine with all the *nodes* in the CHART to form a new *nodes*. But the problem here is how to combine.

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

*The found 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 found grammar rule is not a Chomsky-form;* it has more than two elements on the right part. In this case, the parser uses a virtual node with the *wait* parameter which denotes the lack part to complete the rule. It means that when A *node* and B *node* 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 start-end position, two *nodes* will combine together and form the *node* (E, wait=“D”).

After the parsing process ends, if the *node* (S, 1, n, wait=“”) is founded in CHART, the parsing process is successful. If not, the parsing process is failed.

The virtual node method can solve the problem of combination (how to combine) but the cost of this method is too expensive to deal with. Because of the huge of the grammar rule set (approximately over 10000 rules!!), the combination using virtual node method will generate a large quantity of redundant *nodes.* Specified as table 1, the combination of two instance elements has formed so many new elements, and only few of them are significant.

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")* |

## The PHM model

### The basic idea

Instead of using virtual node like virtual node method, PHM uses *combinable chains* to form new *nodes* 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 previous *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 PHM model, all the combinable chains of a candidate *node* and all the *nodes* in CHART will be checked whether it is the right part of any rule in the syntactic grammar rules or not? All the satisfied chains will form new *nodes* using the relevant grammar rule. Unlike virtual node method, PHM model does not form the redundant virtual *nodes* and it decreases the number of loop steps of A\* algorithm.

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

1. the *nodes in* CHART

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X1  (1-8) | X2  (6-16) | X3  (15-35) | X4  (5-20) | X5  (2-7) | X6  (10-11) |
| X7  (8-27) | X8  (2-21) | X9  (9-11) | X10  (2-13) | X11  (6-14) | X12  (15-26) |
| X13  (14-23) | X14  (5-18) | X15  (1-7) | X16  (9-16) | X17  (12-17) | X18  (7-18) |
| X19  (6-25) | X20  (13-26) | X21  (11-16) | X22  (9-24) | X23  (11-20) | X24  (8-18) |
| X25  (7-16) | X26  (14-16) | X27  (4-6) | X28  (13-21) | X29  (4-8) | X30  (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:

1. all the combinable chains of the candidate with CHART *nodes.*

|  |  |  |
| --- | --- | --- |
| 1 | Position | *node* |
| 2 | [2-7] **[7-10]** | X5**X** |
| 3 | [1-7] **[7-10]** | X15**X** |
| 4 | [1-7] **[7-10]** [10-11] [11-13] | X15 **X** X6 X30 |
| 5 | [2-7]**[7-10]**[10-11] [11-13] [13-26] | X5 **X** X6 X30 X20 |
| 6 | **[7-10]** [10-11] [11-20] | **X** X6 X23 |
| 7 | **[7-10]** [10-11] [11-13] | **X** X6 X30 |
| 8 | **[7-10]** [10-11] [11-13] [13-26] | **X** X6 X30 X20 |
| 9 | **[7-10]** [10-11] [11-13] [13-21] | **X** X6 X30 X28 |
| 10 | [2-7] **[7-10]** [10-11] | X5 **X** X6 |
| 11 | [2-7] **[7-10]** [10-11] [11-16] | X5 **X** X6 X21 |
| 12 | [2-7] **[7-10]** [10-11] [11-20] | X5 **X** X6 X23 |
| 13 | [2-7] **[7-10]** [10-11] [11-13] | X5 **X** X6 X30 |
| 14 | **[7-10]** [10-11] [11-16] | **X** X6 X21 |
| 15 | [1-7] **[7-10]** [10-11] | X15 **X** X6 |
| 16 | [2-7] **[7-10]** [10-11] [11-13] [13-21] | X5 **X** X6 X30 X28 |
| 17 | [1-7] **[7-10]** [10-11] [11-16] | X15 **X** X6 X21 |
| 18 | [1-7] **[7-10]** [10-11] [11-20] | X15 **X** X6 X23 |
| 19 | **[7-10]** [10-11] | **X** X6 |
| 20 | [1-7] **[7-10]** [10-11] [11-13] [13-26] | X15 **X** X6 X30 X20 |
| 21 | [1-7] **[7-10]** [10-11] [11-13] [13-21] | X15 **X** X6 X30 X28 |

Thus, assuming that there is a rule as (A → X5 **X** X6 X30 X28)relevant to the 16th chain in the table 3, the *node* A(2,21) will be formed and will be added to AGENDA.

### PHM combinable-chains generator model

PHM combinable-chains generator model includes two phases: classification phase and combinable chains generation phase.

Classification phase (CP)(1): the parser classifies the nodes of CHART into the difference blocks.

Combinable chains generation phase (CGP)(2): the parser generates all the combinable chains and uses them to create a new node which is added into AGENDA.

#### Classification phase

The PHM classification phase is based on *pigeon hole sort* algorithm idea. There are holes which will be created for adding pigeon. But the holes in PHM are used for CGP(2) instead of sorting.

The holes in the PHM are divided into two types: the *left hole*s and the *right hole*s (Figure 1). Let assuming that X is a candidate *node*.

We have two kind of *node* in CHART:

* *Left node of X*: This is a set of *nodes* that have their *end* position <= *start* position of X. All the *nodes* which have the same *end* position will be grouped in a block labeled as *end* position of them*.* And a set of all these blocks is labeled as left holes.
* *Right node of X*: This is a set of *nodes* that have their *start* position >= *end* position of X. All the *nodes* that have the same *start* position will be grouped in a block labeled as *start* position of them*.* And a set of these blocks is called as right holes.

#### Combinable chains generation phase

With the input as the classified CHART, the parsing system begins generating the combinable chains. This phase includes three main parts: “generating *left chain*s”, “generating *right chain*s” and “*generating combinable chains*”.

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

* Parsing system processes the X *node*, save the left combinable chain corresponding to X and get the S(X) from *left holes*.
* 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 holes to access*”.

This part resembles to th*e tree traversal* at X root. When the *node* Y is processed, *a corresponding* combinable chain is the path from Y to *X on* the tree. For example, for the E node, a relevant generated left 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 chain*s.

***2. Generating right chains****:* the same progress as the “generating *left chain*s” 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 part can be described as:

* Parsing system processes the X *node*, save the right combinable chain corresponding to X and get the E(X) from *right holes*.
* The same with *left chain*s generation part, this progress is done recursively for all the *nodes* in the E(X) until it satisfies “*there is not any block in right holes to access*” condition.

This part resembles 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 major parts, we got a set of combinable chains of candidate *node* X with CHART in each loop step of A\* parsing algorithm using PHM model.

## Pruning map in PHM model

As mentioned above, PHM model is proposed in order to improving the speed of parsing system, to reduce the number of new candidate *nodes* of the A\* algorithm in each loop step. However, the PHM model is still not optimal because of the processing time for each loop step. A number of new candidate *nodes* of A\* using PHM model is less than the A\* using virtual node method but the processing time for each loop step of the PHM 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 PHM model, we found that there are approximately 8% of the combinable chains that could be used. Therefore, the virtual node method and PHM model have the same problem: redundancy. The virtual node method got the redundancy of the number of new candidate *nodes* in each loop step, but the A\* using PHM model got the redundancy of the number of combinable chains in each loop step. Because of this redundancy, PHM model is not only slower than virtual node algorithm in some case, but also got stuck 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, PHM model use a pruning map. Instead of processing all the combinable chains, the parser will use this map to 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 reduce the complex of parsing process.

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

Rchain – A set of syntactic grammar rules, each rule contains “chain” in the right part.

Fchain – 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. PHM basic model only uses the *node* position to generate chain, but the *node* tag is not used.

For instance, if we have two nodes: NP(1,7) and PP(1,7). In the basic PHM model, they are just the same, even their tag is difference. So, a pruning map in PHM model will show you how to use the information of *tag* to reduce the processing time of PHM.

Through the analysis above, the tag of *node* is also very important in the combinable chain generating process. So the pruning map of PHM model will use the information about the tag of the node in grammar rule to improve the performance of system.

The algorithm using pruning map in PHM model includes two phases:

* *Statistic training phase*: using the set of syntactic grammar rule to train and create a pruning map.
* *Pruning phase*: in PHM combinable chain generation phase, use the pruning map which was trained from data to prune the bad candidate.

### Statistic training phase

The training data of the PHM pruning map is the set of syntactic grammar rules.

Specifically, with each POS in the set of syntax grammar rules, 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.

#### 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-tag*(A). Similarly, the H and I tags are the tags that are adjacent to the left of B in the set R*left-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.

#### 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-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 PHM.

### Pruning phase

With the input is the classified CHART; PHM combinable chain generation phase will perform normal with the support of PHM pruning map.

The pruning map 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”. PHM processor will use these results to create new nodes immediately.For instance, the Figure 7 presents the *left chain* generator of PHM model. 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.

# experiment and result

This section presents the preparation and the result of experiment to illustrate the performance of A\* parsing algorithm using PHM model.

## Preparation for experiment

## Results

# conclusion

# 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*](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