A New Algorithm To Evaluate Terminal Heads Of Length K

Xinghuang Xu
Department of EECS
Wichita State University
xxxu3@wichita.edu

ABSTRACT

Finding the terminal heads of length k of a given string in a context-free language has proven to be essential in the design of LL(k) and LR(k) parser generators. Improvement on this technique would greatly enhance of performance of LL/LR parser generators. Aho and Ullman have proposed a method $FIRST_k(\alpha)$ to toggle this problem. This paper presents an innovative alternative to $FIRST_k(\alpha)$ called THREAD. We conduct performance evaluation and conclude that our method performs better under some scenarios such as when the string is long.

CCS Concepts

•**Software and its engineering** → **General programming languages;** *Translator writing systems and compiler generators;* •**Theory of computation** → *Grammars and context-free languages;*

Keywords

First$_k(\alpha)$; THEAD$_k(\alpha)$; Terminal heads; LR(k)

1. INTRODUCTION

1.1 Overview

The algorithm that finds the first k terminal heads of a given string in a context-free grammar is denoted as $FIRST_k(\alpha)$ in this paper. The construction of both top-down(LL) and bottom-up(LR) parsers is aided by $FIRST_k(\alpha)$. During LL/LR parsing, $FIRST_k(\alpha)$ allows us to choose which production to apply, based on the next input symbol. The most widely used case is the computation of $FIRST_k(\alpha)$ when k = 1, because LALR(1), LL(1), SLR(1) and LR(1) are the most efficient algorithms in parsing programming language grammars. On the other hand, when k>1, algorithms like LL(k) with a lookahead of more than 1 tokens are more complex but they can often handle more complex and certain ambiguous context free grammars and are often used in language translation and natural language processing.

The most popular real world parser generator solutions are Bison and ANTlR. Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables[10, 9, 2]. ANTIR on the other hand is a LL(k) parser generator[13, 1]. Even though Bison and ANTIR undertake different paths in the parsing strategy they choose, both of them computes the $FIRST_k(\alpha)$ set directly or indirectly in their parsing processes. The main contribution of this paper is the proposal of a new innovative approach in computing the $FIRST_k(\alpha)$ sets.

The main contribution of this paper is the proposal for an innovative algorithm, which we call $THEAD_k(\alpha)$, to evaluate the terminal heads of length k of a given string in a context-free grammar. It is an alternative to the $FIRST_k(\alpha)$ algorithm of Aho and Ullman [5], and takes a very different approach. In this paper, we will present the algorithm, give examples, compare to the method of Aho and Ullman, and discuss other related issues.

Since $FIRST_k(\alpha)$ is a fundamental algorithm that works as a basic building block in compiler theory and practice, its improvement should have wide impact.

In the rest of this section, we will present the notation conventions and the problem definition. After which, we will survey related works on computing the FIRST set in section 2. In section 3, we will present our algorithm denoted as THREAD along with an easy to understand example. The correctness and complexity of the algorithm will also be covered in Section 3. After section 3, we will compare THREAD with other algorithms in section 4. Lastly, we conclude the whole paper in Section 5.

1.2 Notation Conventions

Alphabet: A set of symbols, where a symbol is a non-divisible basic element of the alphabet.

String: A sequence of symbols concatenated together. We represent the length of a string s as $\mid s \mid$. A string is said to *vanish* if it can derive the empty string. We use Greek letters $\alpha, \beta, \gamma, \ldots$ to represent strings. An empty string is represented by ϵ.

(G)rammar: A *grammar* for a language L is defined as a 4-tuple $G = (N, \Sigma, P, S)$.

N: A set of non-terminal symbols. A non-terminal symbol can appear on either the left or right side of productions. We use upper case Roman letters $A, B, C, \ldots$ to represent non-terminals.

Σ: A set of terminal symbols disjoint from the set N. A

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

terminal symbol appears only on the right side of productions. We use lower case Roman letters $a, b, c, \ldots$ to represent terminals

P: A set of productions.

S: The start symbol from which the production rules originate from.

$k-head$: A $k-head$ of a string S is a string which is made of the first k symbols of S, or the first k symbols of any string that can derive from S.

$k-terminalhead$ or $k-thead$: A $k-head$ of S which is made up of terminal strings only.

EXAMPLE 1.1. *Given grammar G1:*
$X \to XY \mid a,\ Y \to b \mid \epsilon$

Here a and b are terminal symbols because they appear only on the right side of the productions of G1. X and Y are non-terminal symbols because they can appear on the left side of the productions of G1. Y vanishes because it can derive the empty string. The shortest string X can derive is a, therefore it does not vanish because it cannot derive the empty string.

Given string $\alpha = XY$, its 1-head can be X or a, and its 2- head can be XY, XX, aY, Xb, ab or aa. Its 1-thead is a, and its 2-thead can be aa or ab.

1.3 Problem Definition

The problem of finding the terminal heads of length k given a string α is equivalent to finding $FIRST_k(\alpha)$ given a grammar G and string α.

For a CFG $G = (N, \Sigma, P, S)$,

$FIRST_k(\alpha) = \{x| \quad \alpha \underset{lm}{\overset{\star}{\Longrightarrow}} x\beta \quad and \quad |x| = k \quad or \quad \alpha \overset{\star}{\Rightarrow} x \quad and \quad |x| < k\}$

Where $\underset{lm}{\overset{\star}{\Longrightarrow}}$ means 0 or more steps of leftmost derivatives. In short, $FIRST_k(\alpha)$ consists of all terminal prefixes of length k (or less if α derives a terminal string of length less than k) of the terminal strings that can be derived from α. Following are two examples on the calculation of $\text{FIRST}_k(\alpha)$.

EXAMPLE 1.2. *Given grammar $S \to NM, N \to st, M \to bc$. We want to find $FIRST_k(\alpha)$ for string $\alpha = NM$. This is a trivial case, we can just plug N and M into NM to obtain $\alpha = stbc$. Calculation of $FIRST_k(\alpha)$ is easy: $FIRST_1(\alpha) = s$, $FIRST_2(\alpha) = st$, $FIRST_3(\alpha) = stb$, $FIRST_4(\alpha) = stbc$.*

EXAMPLE 1.3. *Given grammar $S \to NML$, where $N \to Ns \mid \epsilon$, $M \to Mt \mid \epsilon$. Here ϵ is the empty string. We want to find $FIRST_k(\alpha)$ for string $\alpha = NML$.*

Then actually $N = s^*$, and $M = t^*$, and $\alpha = s^*t^*bc$.

$FIRST_1(\alpha) = \{s, t, b\}$,

$FIRST_2(\alpha) = \{ss, st, sb, tt, tb, bc\}$,

$FIRST_3(\alpha) = \{sss, sst, ssb, stt, stb, sbc, ttt, ttb, tbc\}$,

$FIRST_4(\alpha) = \{ssss, ssst, sssb, sstt, sstb, ssbc, sttt, sttb, stbc, tttt, tttb, ttbc\}$.

2. RELATED WORK

We did a survey on the work related to the calculation of FIRST set and found the following literatures.

2.1 Computing Lookahead Sets

Lookahead set can be defined in terms of FIRST and FOLLOW sets. An efficient computation of FIRST set would lead to an efficient way to compute lookahead sets.

The work of DeRemer and Pennello[8] tries to provide an efficient way to compute the LALR(1) lookahead sets in time linear in the size of the relations. On the other hand, Madsen and Kristensen[11] work on methods for computing LALR(k) lookahead sets.

The work of Kristensen and Madsen [11] discussed computing $FIRST_k$ for finding lookahead strings, which is needed by their LALR(k) algorithm. Their method is based on simulating all steps involved in parsing starting from a relevant state in a LR(0) machine. Given an example of calculating $LALR_k$ for $[A \to .\alpha]$, their method wants to obtain the sets $\cup\{FIRST_k(\psi_i) \mid i = 1, 2, , n\}$for all items $[B_i \to \varphi_i . A\psi_i]$, which *"may be computed by simulating all possible steps that the parse algorithm may take starting in the state $GOTO_k(S, A)$ with an empty parse stack."* They further pointed out that the set $\cup FIRST_k(\psi_i)$ is not enough, and proceeded to discuss how to cover edgy cases such as when the grammar is circular or contains ϵ-productions, and ended their discussion with cases where the simulated parsing might fail due to circularity.

The PhD thesis of Parr [13] proposed a method to compute $FIRST_k(\alpha)$. This is used in the implementation of LL(k) parser generator ANTLR. Parr's PhD thesis introduces the GLA grammar representation in chapter 3, and explains lookahead computation and representation in chapter 4. Basically, a data structure called GLA (Grammar Lookahead Automata) is used to represent grammars. To calculate LR(k) lookahead, do a constrained walk of a GLA, and the lookaheads are stored as a lookahead DFA (Deterministic Finite Automata). He also discussed how to solve the cycle issue with cache mechanism. This is similar to the method of Kristensen and Madsen in that it utilizes the parsing machine to do the computation and tightly integrates the calculation of lookahead strings with parsing, and in that none of them is a standalone method to calculate $FIRST_k(\alpha)$.

All the above work redefined LOOK-AHEAD to their favor to avoid the direct computation of FIRST and FOLLOW sets. They leverage the parsing machine for lookahead set computation and tightly integrate the lookahead set computation with parsing. In summary, they did not provide a standalone solution to compute FIRST set.

2.2 Method of Aho and Ullman

The only stand alone algorithm we were able to find comes from Aho and Ullman[5].

They first define an operator $\oplus_k$ such that: given an alphabet Σ and two sets $A \subseteqq \Sigma^*$, $B \subseteqq \Sigma^*$, $S = A \oplus_k B$ is the set of all strings formed from the ordered concatenation of string pairs (a, b), where $a \in A$, $b \in B$, and the length of strings in S is less than or equal to k.

Now given a context free grammar $G = (N, \Sigma, P, S)$ and a string $\alpha = X_1X_2X_n$ in $(N \cup \Sigma)*$, $FIRST_k(\alpha) = FIRST_k(X_1) \oplus_k FIRST_k(X_2) \oplus_k \ldots \oplus_k FIRST_k(X_n)$, the prove can be found in the book. From this point on, we only need to calculate $FIRST_k(X)$ for any X.

The following steps are used to calculate $FIRST_k(X)$: Define a sets $Fi(X)$ for X in $N \cup \Sigma$ and for increasing values of i, $i \geq 0$, as follow:

1. $F_i(X) = \{a\}$ for all a in Σ and $i \geq 0$;
2. $F_0(X) = \{x \in \Sigma^{\star k} | A \to x\alpha$ is in P, where either |x|=k or $|x| < kand\alpha = e\}$

3. Suppose that $F_0, F_1, \ldots, F_{i-1}$ have been defined for all A in N. Then $F_i(A) = \{x|A \rightarrow Y_1, \ldots, Y_n$ is in P and x is in $F_i(Y_1) \oplus_k F_i(Y_2) \oplus_k \ldots \oplus_k F_i(Y_n)\} \cup F_{i-1}(X)$

4. As $F_{i-1}(A) \subseteq F_i(A) \subseteq \Sigma^{\star k}$ for all A and i, eventually we must reach an i for which $F_{i-1}(A) = F_i(A)$ for all A in N. Let $FIRST_k(A) = F_i(A)$ for that value of i.

In summary, the method of Aho and Ullman breaks down the task of evaluating the terminal heads of length k of a string α into applying the $\oplus_k$ operation on the component symbols of α. It solves the second problem by building a table from bottom up like in dynamic programming.

3. THE NEW ALGORITHM

In this section we propose the new algorithm [12] with a simple example, prove the correctness and complexity of the new algorithm.

3.1 The $THEAD_k(\alpha)$ Algorithm

We use $THEAD_k(\alpha)$ as the name of the new algorithm, and also use it to represent the set of terminal heads of string α, where the length of each terminal head string is k. $THEAD_k(\alpha)$ and $FIRST_k(\alpha)$ are equivalent.

To illustrate the algorithm, we define the following notations:

For a string $\alpha = X_1 X_2 \ldots X_n$, $|\alpha|$ is the length of α ($|\alpha|$ = n); $\alpha[i]$ is the ith symbol of string α; $h(\alpha, k)$ denotes the first k symbols of α, i.e., prefix string of α of length k; $h_v(\alpha, k)$ is a substring of α that consists of the prefix string of α up to the k-th symbol that does not vanish, or the entire α string if it contains less than k symbols that do not vanish; $prod(\alpha, i)$ is the set of strings obtained by applying all possible productions to the ith symbol X_i of α.

We also let T stand for the set of *Terminals*, and NT stand for the set of *Non-Terminals*. T_k stands for the set of strings made of Terminals and whose length is k. $\emptyset$ stands for the empty set.

Algorithm 1, $THEAD_k(\alpha)$, is shown in Figure 1.

In Algorithm 1, **H** and **S** are sets of strings initially empty. **H** is used to hold all k-THEADs and **S** contains all m-THEADS where m<k. **L** is an auxiliary ordered list of strings which initially consists just of $h_v(\alpha, k)$.

The following steps are repeated to get i-THEADS where goes from 1 to k:

1. Expansion of Strings in L using $prod(\alpha, i)$(Lines 5-10): add to the end of L the result of applying all possible productions to the i^{th} symbol in the current member β of L, omitting strings that are already in L, and truncating all members added which have k or more symbols that do not vanish, by deleting the part of the string following the k-th symbol that does not vanish.

2. Filter Invalid Strings in L(Lines 11-13): remove from L all strings whose ith symbol is a non-terminal. After this step, L should only contain strings whose first i symbols are non-terminals.

3. Extract k-THEADS from L into H(Lines 14-19): remove from L all strings whose prefix of length k consisting entirely of terminals, and add the prefixes of length k involved to the set H.

ALGORITHM 1. $THEAD_k(\alpha)$

INPUT: STRING $\alpha = X_1 X_2 \ldots X_n$; Integer k: length of theads.
OUTPUT: SET H – CONTAINS K-THEADS OF α, AND (OPTIONALLY) SET S – CONTAINS M-THEADS OF α, M < K.

```
1   H ← Ø
2   S ← Ø
3   L ← { h_v(α, k) }
4   for i = 1 to k do
5       foreach string β in L do
6           φ = prod(β, i)
7           foreach string γ in φ do
8               L ← L ∪ { h_v(γ, k) }
9           end foreach
10      end foreach
11      foreach string β in L do
12          if β[i] ∈ NT then L ← L − { β }
13      end foreach
14      foreach string β in L do
15          if h(β, k) ∈ T^k then
16              L ← L − { β }
17              H ← H ∪ { h(β, k) }
18          end if
19      end foreach
20      foreach string β in L do
21          if |β| < k AND β ∈ T^|β| then
22              L ← L − { β }
23              S ← S ∪ { β }
24          end if
25      end foreach
26      if L == Ø then stop
27  end for
```

Figure 1: Algorithm $THREAD_k(\alpha)$

4. Extract m-THEADS ($m < k$) from L into S(Lines 20-25): remove from L all strings of length less than k which consist entirely of terminals, and add these to the set S.

5. Early termination(line 26), if L is empty, the algorithm terminates.

After the termination of the outermost for loop, **H** will contain the required set of terminal strings of length k of α, i.e., the k-head set of α; and S will contain the set of terminal strings of length less than k which are derived from α. Obviously, **H** gives the result of $THEAD_k(\alpha)$.

The entire algorithm derives a closure of the initial string in L, where each derived string in the closure satisfies the requirements on the length (should be equal to k) of the strings, and on the type of symbols (should be terminal symbol) in the strings.

3.2 An Example of $THEAD_k(\alpha)$

In this section we show how $THEAD_k(\alpha)$ and $FIRST_k(\alpha)$ work on the same input string.

EXAMPLE 3.1. *Given grammar G2 (ϵ is the empty string):*
$X \rightarrow XY|a|\epsilon$
$Y \rightarrow Z|y|\epsilon$

$Z \to X|z|\epsilon$
$U \to u$
Find the set of 2-theads of XYZU using Algorithm 1: $THEAD_k(\alpha)$.

Table 1: Example 4, roud 1 (i=1)

i	j	String added to L	String Sequence Number
1	1	XYZU	1
		YYZU	2
		xYZU	3
		YZU	4
	2	ZYZU	5
		yYZU	6
	3		
	4	ZZU	7
		yZU	8
		ZU	9
	5	zYZU	10
	6		
	7	XZU	11
		zZU	12
	8		
	9	XU	13
		zU	14
		U	15
	10		
	11	xZU	16
	12		
	13	YU	17
		xU	18
	14		
	15	u	19
	16		
	17	yU	20
	18		
	19		
	20		

Since symbols X, Y and Z can all vanish, and U does not vanish, the string XYZU contains less than 2 symbols (i.e., only 1) that do not vanish, therefore we need to include the entire string XYZU as the initial element in the list L. Thus, at the beginning, L = XYZU.

First round of operation for i = 1 is shown in Table 1.

At this time, the step of lines 5-10 finishes. Next we follow lines 11-25. Remove from L all strings with nonterminals in the ith (first) position; remove from L all strings whose prefixes of length 2 consisting entirely of terminals, and add these prefixes to H; and remove from L all strings of length less than 2 and contains only terminal strings. At this time, we have H = {}, S = {u}, L = {xYZU, yYZU, yZU, zYZU, zZU, zU, xZU, xU, yU}.

The second round where i = 2 is shown in Table 2.

Remove all strings with non-terminals in the ith (second) position, remove all strings whose prefixes of length 2 are made up of terminals, and remove all strings of length less than 2 and contains only terminal strings, we have H = {xy, yy, zy, zz, zu, xu, xz, yz, yx, yu, zx, xx}, S = {u}, L = {}.

EXAMPLE 3.2. *Given grammar G2 as in Example 3.1, find the set of 2-theads of XYZU, this time use the $FIRST_k(\alpha)$ algorithm of Aho and Ullman.*

Table 2: Example 4, round 2 (i = 2)

i	j	String added to L	String Sequence Number
2		xYZU	1
		yYZU	2
		yZU	3
		zYZU	4
		zZU	5
		zU	6
		xZU	7
		xU	8
		yU	9
	1	xZZU	10
		xy	11
	2	yZZU	12
		yy	13
	3	yXU	14
		yz	15
	4	zZZU	16
		zy	17
	5	zXU	18
		zz	19
	6	zu	20
	7	xXU	18
		xz	22
	8	xu	23
	9	yu	24
	10	xXZU	25
	11		
	12	yXZU	25
	13		
	14	yYu	27
		yx	28
	15		
	16	zXZU	29
	17		
	18	zYU	30
		zx	31
	19		
	20		
	21	xYU	32
		xx	33
	22		

Following the steps in Aho and Ullman's algorithm, we need $FIRST_k(\alpha)$, where α = XYZU, and k = 2.

$F_i(p) = \{p\}$, for all $p \in x, y, z, u, \epsilon$, and $i \geq 0$.
$F_0(X) = \{x, \epsilon\}$
$F_0(Y) = \{y, \epsilon\}$
$F_0(Z) = \{z, \epsilon\}$
$F_0(U) = \{u\}$
$F_1(X) = \{x, y, \epsilon\}$
$F_1(Y) = \{y, z, \epsilon\}$
$F_1(Z) = \{z, x, \epsilon\}$
$F_1(U) = \{u\}$
$F_2(X) = \{x, y, z, \epsilon\}$
$F_2(Y) = \{x, y, z, \epsilon\}$
$F_2(Z) = \{x, y, z, \epsilon\}$
$F_2(U) = \{u\}$

From this point on $F_i(S) = F_2(S)$ for $i \geq 3$, S = X, Y, Z,

U. It converges here. Therefore:
$FIRST_2(X) = F_2(X) = \{x, y, z, \epsilon\}$
$FIRST_2(Y) = F_2(Y) = \{x, y, z, \epsilon\}$
$FIRST_2(Z) = F_2(Z) = \{x, y, z, \epsilon\}$
$FIRST_2(U) = F_2(U) = \{u\}$

Note that here $FIRST_2(X)$ contains strings of length less than 2, because we need to keep them in the intermediate steps, as discussed at the end of section 2.4.

Finally, we can calculate $FIRST_k(\alpha) = FIRST_2(XYZU)$ $= FIRST2(X) \oplus_2 FIRST_2(Y) \oplus_2 FIRST2(Z) \oplus_2 FIRST_2(U)$ $= x, y, z, \oplus_2 x, y, z, \oplus_2 x, y, z, \oplus 2u$ = {xx, xy, xz, xu, yx, yy, yz, yu, zx, zy, zz, zu, u}

We remove strings whose length are less than 2, which is 'u', and obtain {xx, xy, xz, xu, yx, yy, yz, yu, zx, zy, zz, zu}. This is the same result as using our algorithm.

3.3 Proof of Correctness

The prove of the correctness of Algorithm 1 is as follow.

LEMMA 1. *In Algorithm 1, at the end of the i^{th} outer loop cycle (lines 4-27), for each string s in list L, where s $= X_1X_2X_n$, the first i symbols $X1, X2, , Xi$ of s (or all the symbols of s if $\mid s \mid < i$) are terminals.*

PROOF. Prove by induction. For outer loop cycle i = 1, the step of lines 11-13 removes from L all strings whose 1^{st} symbol is a non-terminal. Thus for all the strings remained in L, the 1^{st} symbol is terminal. Now assume at cycle i $= n - 1$, for all the strings in L, the first i symbols are terminals. At cycle i = n, the inner loop (lines 5-10) only makes derivations on the nth symbol, and does not introduce any nonterminal symbols to the first $n - 1$ symbols; next, Algorithm 1 removes from L those strings whose nth symbol is a nonterminal (lines 11-13), thus for all the symbols in L, now their first n symbols are terminals. The remaining steps (lines 14-26) do not alter this fact. Therefore Lemma 1 holds. □

LEMMA 2. *In Algorithm 1, at the end of the i^{th} outer loop cycle, all the possible combinations of i-thead derivations are generated by the inner loop (lines 5-10).*

PROOF. This also can be proved by induction. When i = 1, this is obvious from the inner loop. Assume this holds for i = $n - 1$. When i = n, for each string s in L, the first $n - 1$ symbols of s are all terminals. In the inner loop, for each string s in L, all the possible productions are applied to the n^{th} symbol of s, thus all the possible terminal and non-terminal symbols at the nth position are generated by string s and included in L. These form new derived strings, appended to the end of L, and processed by the next cycle. Thus Lemma 2 holds. □

LEMMA 3. *Algorithm 1 ends in k or less outer loop cycles (lines 4-27) when L becomes empty.*

PROOF. From Lemma 1, for all the strings generated in the k^{th} outer loop cycle, their first k symbols are all terminals, these are then removed from L (lines 14-25). In the cycles, all members added to L that have k or more symbols that do not vanish will be truncated (lines 3, 8 and 12). Thus L will be empty at the end of at most the kth loop cycle, and Algorithm 1 ends. □

THEOREM 1. *When Algorithm 1 ends, all the possible kt-head derivations are included in H, and all m-thead derivations are included in S, where $m < k$.*

PROOF. This follows from Lemma 1, Lemma 2 and Lemma 3. □

3.4 Complexity Analysis

In Algorithm 1, the complexity of the step of lines 6-9 is $O(|P_{ij}|)$, where $|P_{ij}|$ is the number of possible productions to the i^{th} symbol in the jth member of L. For the loop of lines 5-10, the complexity is $O(|P_{ij}||L|)$.

The complexity of the entire algorithm is hard to analyze directly, but it is easy to see that, since the primary output is set H, the theoretical lower boundary of the number of steps needed is equal to the number of elements in the output set: $\Omega(|H|)$. H is the set of terminal strings of length k of α, so $\Omega(|H|) = \Omega(|T|_k)$, where |T| is the number of terminals in the alphabet. This is the theoretical lower boundary of both time and space requirements. Obviously, it is exponential in nature as expected.

3.5 Application

There could be many application of our new algorithm. One of which is in the implementation of LR(k) parser generator. One of the major calculation of LR(k) algorithm is the computation of the lookahead set. We designed and implemented the Edge-Pushing LR(k) algorithm, which depends on the $THEAD_k(\alpha)$ function to calculate k-lookahead. The algorithm is implemented in the HYACC parser generator, which is available as an open source parser generator[7, 3]. For more detail of the Edge-Pushing algorithm, please go to the Appendix.A for a full version of the algorithm, which is taken from a previous paper [6]. The $THEAD_k(\alpha)$ algorithm is being used on line 13.

4. COMPARISON WITH OTHER ALGORITHMS

4.1 Theoretical Comparison

Aho and Ullman's method and our method are both standalone algorithms to compute $FIRST_k(\alpha)$, where the computation rely on a set of production rules of the grammar only, and the parsing machine is not needed. Thus these two methods are more comparable.

Aho and Ullman's method takes a bottom up approach by first calculating $FIRST_i(X)$ for each symbol X, $i = 1, 2, \ldots k$, then combining these building blocks to obtain $FIRST_k(\alpha)$. This is a systematic approach, which is also demonstrated in their handling of $FIRST_1(\alpha)$, which is discussed in [4]. Once the preparation phase is done, for whatever input string, the task boils down to applying the $\oplus_k$ operation on the consisting symbols of the input string, which concatenates elements from each set. However, the systematic nature also means that the overhead must always be taken to achieve good efficiency. From a practical point of view, since input strings are unknown, the entire preparation step must be done and its result be cached for later use.

In comparison, our method takes a top down approach. No previous computation is needed. The algorithm computes $FIRST_k(\alpha)$ on the fly based on symbols included in the input string. No cache is needed. It removes unnecessary overhead strings on the way of computation. In nature, both methods are equivalent. Our method can also be used for the preparation process of Aho and Ullman's method.

Another difference is that the $FIRST_k(\alpha)$ method of Aho and Ullman gives a set of terminal heads whose length $L \leq k$, and this set must be kept during the entire calculation process, only at the very end can we remove those $L < k$. In comparison, our method separates terminal heads into two sets, for one set the length of terminal heads $L = k$, and for the other set $L < k$. The second set where $L < k$ can be ignored from the calculation process.

4.2 Performance Comparison

4.2.1 Implementation Detail

We implemented both the $THEAD_k(\alpha)$ algorithm and the $FIRST_k(\alpha)$ algorithm in ANSI C from scratch, and compared their performance. To make the comparison of the two algorithms reasonable, it is necessary to implement them with similar data structures.

The major operations involved in both algorithms are set operations. In the current implementation, a set is implemented as a linked list. Search in the set is done by going through the list in linear order. That a linked list is chosen for the implementation is because of the nature of the $THEAD_k(\alpha)$ algorithm: a new generated string has to be appended to the end of the current set, which makes queue a natural and necessary choice. A queue of unknown size as in the current scenario is in turn naturally implemented as a linked list.

To guarantee similar search experience for both algorithms, an ordered list is used. For the method of Aho and Ullman, this is no problem. But for the $THEAD_k(\alpha)$ method, the queue (implemented as a list) to be appended that can not be ordered, so an auxiliary list is provided which stores the same strings as the queue but is in sorted order, such that when a search in the auxiliary ordered list does not return a hit, the new string is appended to the end of the queue. The maintenance of two lists in the $THEAD_k(\alpha)$ algorithm implementation obviously will slow it down to some degree.

This implementation can be improved by providing an auxiliary binary search tree or a hash table to both methods, which works much more efficient when decide if a string exists in a set. This improvement should be of more significance to the performance of the $THEAD_k(\alpha)$ algorithm implementation according to the above discussion.

Finally, a linked list suffices for all the operations of the $THEAD_k(\alpha)$ algorithm. For the algorithm of Aho and Ullman, an array is also used to store the pre-computed $FIRST_k(X_i)$ values of all the symbols X_i, such that given a random string $= X_1X_2 \ldots X_n, FIRST_k(X_i)$ can be retrieved in constant time using index of X_i in the symbol table for the calculation of $FIRST_k(\alpha) = FIRST_k(X_1) \oplus_k FIRST_k(X_2) \oplus_k \oplus_k FIRST_k(X_n)$.

4.2.2 Experiment

In each experiment, the start time and end time are measured multiple times, and then average start time is subtracted from average end time to obtain the running time. The study was conducted on a Sun Microsystems sun4u Netra 440 server running Solaris. CPU is 1.6GHz, memory is 12 GB. For all the experiments below, test case 2 uses the most memory (hundreds of MB), so memory is not an issue. In the figure legends, $THEAD$ represents $THEAD_k(\alpha)$, and $FIRST$ represents $FIRST_k(\alpha)$.

Grammar G2 is used as the testing grammar.

TEST CASE 1. α = ***UUUUUUUUUU and k = 1 to 10*** *Result is shown in Table 3 and Figure 2. When α = UUUUUUUUUU, there is only one terminal head, which is uk for k = 1 to 10. The speed is very fast, at the level of microsecond. The relatively long delay when k = 1 for the $FIRST_k(\alpha)$ algorithm should be caused by the initial construction of the $F_i(X)$ table.*

k	# of k-theads	Time (sec) By HEAD	Time (sec) By FIRST
1	1	0.000022	0.000108
2	1	0.000009	0.000014
3	1	0.000012	0.00002
4	1	0.000018	0.000017
5	1	0.00002	0.00002
6	1	0.000027	0.000026
7	1	0.000032	0.000021
8	1	0.000072	0.000022
9	1	0.000047	0.000026
10	1	0.000053	0.000055

Table 3: Number of generated k-theads and time spent on input string UUUUUUUUUU, for k = 1 to 10

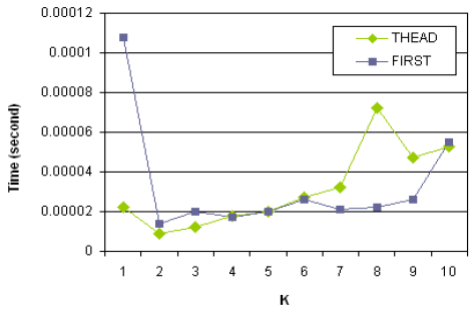

Figure 2: Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ for α = UUUUUUUUUU, k = 1 to 10

TEST CASE 2. α = ***XXXXXXXXXX and k = 1 to 8*** *Result is shown in Table 4 and Figure 3. This is the worst case scenario where the theoretical bound of exponential behaviour is observed. This is because each symbol of the input string is a non-terminal (X), which can derive 3 terminals x, y and z. The number of k-theads that can be generated is 3^k. When k is as small as 10, this will take hours to finish. The result is similar when α = YYYYYYYYYY or α = ZZZZZZZZZZ.*

TEST CASE 3. α= ***XYZUXYZUYX, k = 1 to 10*** *Result is shown in Table 5 and Figure 4. Here α is a randomly generated string. We can see that THEADk(Îś) performs better than FIRSTk(Îś) for k = 1 to 9, but for k = 10, FIRSTk(Îś) runs faster. This possibly has to do with the way of implementation: in the implementation of FIRSTk(Îś), an ordered list is used to store the strings generated intermediately; for THEADk(Îś), the list used cannot be ordered, since new inserted strings will need to be processed and have to be attached to the end. When inserting a new generated string to the end of list L, THEADk(Îś) will search*

k	# of k-theads	Time (sec) By THEAD	Time (sec) By FIRST
1	3	0.000242	0.000221
2	9	0.001302	0.00145
3	27	0.00599	0.009041
4	81	0.032146	0.065045
5	243	0.213318	0.425997
6	729	1.463382	3.282263
7	2187	12.21782	26.23495
8	6561	135.462	297.5679

Table 4: Number of generated k-theads and time spent on input string XXXXXXXXXX, for k = 1 to 8

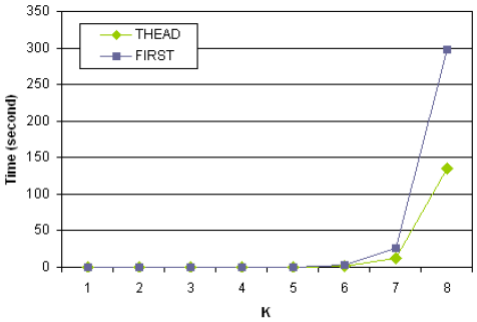

Figure 3: Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ for α = XXXXXXXXXX, k = 1 to 8.

through the entire list to make sure it does not exist yet. To overcome this issue an auxiliary ordered list is used in the implementation. This slows it down when the list is long. Of course, better implementation using more efficient data structure can improve this scenario.

k	# of k-theads	Time (sec) By THEAD	Time (sec) By FIRST
1	4	0.000079	0.000315
2	16	0.00038	0.001807
3	63	0.002083	0.016498
4	162	0.011877	0.063377
5	486	0.100032	0.460147
6	1296	0.624867	2.756787
7	2916	3.3104	11.8662
8	4374	14.64284	26.12881
9	6561	62.89018	71.2255
10	6561	94.49193	81.37379

Table 5: Number of generated k-theads and time spent on input string XYZUXYZUYX, for k = 1 to 10

TEST CASE 4. ***Average on 100 strings of length 10, k = 1 to 8*** *100 input strings, each of length 10, are generated from the alphabet of X, Y, Z, U using a random number generator, and then fed to the algorithms to compare their performance. This means the input strings may be like:*

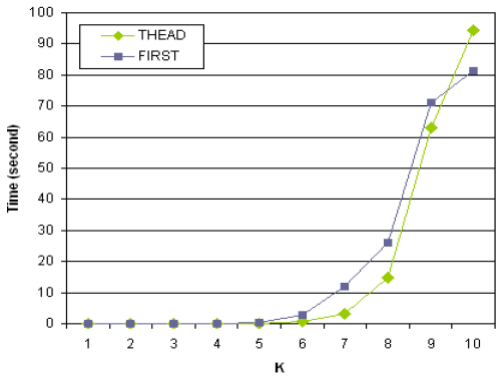

Figure 4: Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ for α = XYZUXYZUYX, k = 1 to 10.

1 YXXXYUUUUU
2 UZZUUUZXXY
3 YZZUYZZYZU
4 ZZUZUZYUZY
....
100 UUYXXUUXUY

Result is shown in Table 6 and Figure 5. Table 6 shows the average number of k-theads generated and average time used by the $THEAD_k(\alpha)$ and $FIRST_k(\alpha)$ algorithms over 100 input strings of length 10, and k = 1 to 8. Figure 5 shows the graphical version of the average time used when k increases. It can be seen that the $THEAD_k(\alpha)$ algorithm uses less time.

k	Avg # of k-theads	Time (sec) By THEAD	Time (sec) By FIRST
1	3.07	0.000068	0.000177
2	10.37	0.000381	0.000969
3	32.73	0.001999	0.006003
4	95.43	0.011761	0.03663
5	270.25	0.078635	0.246849
6	697.89	0.505454	1.496519
7	1662.39	3.484229	8.207717
8	3669.3	27.723004	55.275918

Table 6: Average number of generated k-theads and time spent on 100 random strings of length 10, for k = 1 to 8

TEST CASE 5. ***100 strings of length 1 to 100 with k = 2*** *In this test case, k is fixed, while the input string is a kprefix of the following randomly generated string, where input string length $|\alpha|$ = 1 to 100, i.e., the input strings may be like:*

1 Y
2 YZ
3 YZZ
4 YZZY
....
100 YZZYYXZYYXYZUXYYUYXZUYYUZXUYZZ YYZXXXXXUUUYXYZZYZYZUUXZXZYZXX UZUXYZYYYUYZZZZZUZXZYYYYZYYUXZ ZUYZUZXUY

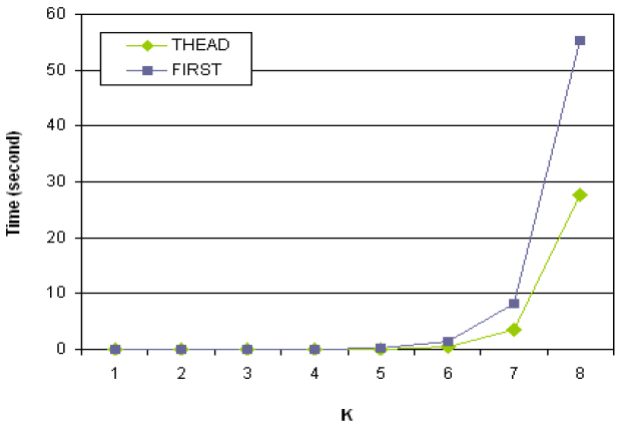

Figure 5: Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ when k increases. Averaged over 100 strings of length 10

Result is shown in Figure 6. The time used by $THEAD_k(\alpha)$ does not increase with k, but it does increase with $FIRST_k(\alpha)$ (and the increase is linear visually from the graph). This is easy to explain. $THEAD_k(\alpha)$ throws away the substring after the second symbol that does not vanish, so each time it starts with the prefix "YZ" of the input string. In comparison, $FIRST_k(\alpha)$ needs to do the $\oplus_k$ operation on every symbol of the input string, and n-1 $\oplus_k$ operations are applied for an input string of n symbols. To overcome this issue, $FIRST_k(\alpha)$ needs to use a preprocessing the same as line 3 of Algorithm 1.

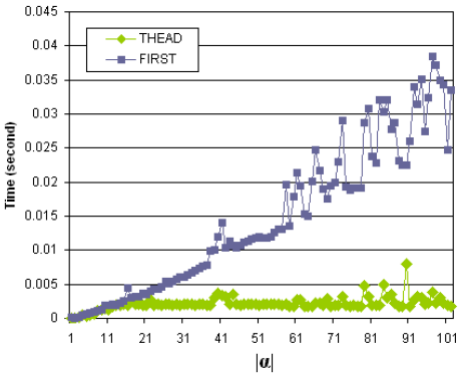

Figure 6: Time cost of $THEAD_k(\alpha)$ versus $FIRST_k(\alpha)$ when k = 2, and string length $|\alpha|$ increases

We can draw several conclusions from the experiments. First of all, when the input string contains terminal symbols only, the speed is the fastest. When the input string contains non-terminal symbols only, the speed is the slowest, and may lead to the worst case scenario: exponential increase in computation time. For a grammar as simple as G2, when k = 10, it will take hours to finish using both algorithms.

In general the $THEAD_k(\alpha)$ algorithm performs better than the $FIRST_k(\alpha)$ algorithm, as shown by test case 4, which is averaged over 100 randomly generated strings of length 10 for k = 1 to 10.

However, it is also possible that $FIRST_k(\alpha)$ runs faster than $THEAD_k(\alpha)$, as shown in test case 3 when k = 10. Finally, when k is small, but input string is long, $THEAD_k(\alpha)$ will perform better than $FIRST_k(\alpha)$, as shown by test case 5. Actually, for this scenario, the time $THEAD_k(\alpha)$ takes will not increase when the size of the input string increase. However, the time used by $FIRST_k(\alpha)$ will increase linearly according to the length of the input string.

5. CONCLUSION

We have provide an introduction to the problem of finding the terminal heads of length k of a string. Earlier solutions to this problem is surveyed. Aho and Ullman's method is the only previously available standalone algorithm for this problem.

We then propose of the new solution $THREAD_k(\alpha)$ with a simple example. The algorithm is further analysed in terms of its correctness, complexity and application. We also pointed out the it has been used to implement the edge-pushing algorithm in the HYACC parser generator

Moreover, we compare the algorithm with Aho and Ullman's solution. We first implemented the two algorithms using comparable data structures then we conduct an empirical study of the two algorithms, $THREAD_k(\alpha)$ and $FIRST_k(\alpha)$. In general, when averaged over a large number of randomly generated input strings, $THREAD_k(\alpha)$ performs faster than $FIRST_k(\alpha)$. When the input string Ĩś is long but k is small, $THREAD_k(\alpha)$ always performs better than $FIRST_k(\alpha)$.

We believe that the improvement of $FIRST_k(\alpha)$ should have wide impact due to the fact that it is a fundamental algorithm that works as a basic building block for many compiler theory.

6. REFERENCES

[1] Antlr. http://www.antlr.org.
[2] Gnu bison. http://www.gnu.org/software/bison.
[3] Lr(1) parser generator hyacc. http://hyacc.sourceforget.net.
[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
[5] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing of *Series in Automatic Computation*. Prentice Hall, Englewood Cliffs, New Jersey, 1972.
[6] X. Chen and D. Pager. The edge-pushing lr(k) algorithm. pages 490–495, July 2011.
[7] X. Chen and D. Pager. Full lr(1) parser generator hyacc and study on the performance of lr(1) algorithms. In *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering*, C3S2E '11, pages 83–92, New York, NY, USA, 2011. ACM.
[8] F. DeRemer and T. Pennello. Efficient computation of lalr(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, Oct. 1982.
[9] C. Donnelly and R. Stallman. *Bison Manual: Using the YACC-compatible Parser Generator*. GNU manual. Free Software Foundation, 2003.
[10] S. C. Johnson. Yacc: Yet another compiler-compiler.

Technical report, Bell Laboratories, Murray Hill, NJ, 1979.

[11] B. B. Kristensen and O. L. Madsen. Methods for computing lalr(k) lookahead. *ACM Trans. Program. Lang. Syst.*, 3(1):60–82, Jan. 1981.

[12] D. Pager. Evaluating terminal heads of length k. Technical report, University of Hawaii, Information and Computer Sciences Department, Nov. 2008.

[13] T. Parr. *Obtaining practical variants of LL(k) and LR(k) for k > 1 by splitting the atomic k-tuple.* PhD thesis, Purdue University, Aug. 1993.

APPENDIX

A. EDGE-PUSHING LR(K) ALGORITHM

Algorithm 2: Edge_Pushing(S)

INPUT: INADEQUATE STATE S
OUTPUT: S WITH CONFLICT RESOLVED, IF S IS LR(K)

```
1  Set_C ← Ø
2  Set_C2 ← Ø
3  k ← 1
4  foreach final configuration T of S do
5      T.z ← 0
6      Let C be the head configuration of T, and X be the
           context generated by C
7      Add triplet (C, X, T) to set Set_C
8  end foreach
9  while Set_C ≠ Ø do
10     k ← k + 1
11     foreach (C:A → α • B β, X, T) in Set_C do
12         k' ← k - C.z
13         calculate ψ ← theads(β, k')
14         foreach context string x in ψ do
15             if x.length == k` then
16                 Insert (S, X, last symbol of string x, C, T) to
                       Set_C2 and add to LR(k) parsing table
17             else if x.length == k` - 1 then
18                 Σ ← lane_tracing(C)
19                 foreach configuration σ in Σ do
20                     σ.z ← C.z + k'
21                     Let m be the generated context symbol in σ
22                     Insert(S, X, m, σ, T) to Set_C2 and add to
                           LR(k) parsing table
23             end if
24         end foreach
25     end foreach
26     Set_C ← Set_C2
27     Set_C2 ← Ø
28 end while
```