Journal of Intelligent & Fuzzy Systems 46 (2024) 2537–2546
DOI:10.3233/JIFS-223021
IOS Press

Statistical analysis of lexemes generated in ‘C’ programming using fuzzy automation

An Empirical approach to NextGen Software

Ranjeet Kaur[a,b,*] and Alka Tripathi[a]
[a]*Department of Mathematics, Jaypee Institute of Information Technology, Noida, Uttar Pradesh, India*
[b]*ABES Engineering College, Ghaziabad, Uttar Pradesh, India*

Abstract. The present work is an effort to support the typographical errors of keywords that are not supported by existing compilers and integrated development environment(IDE) in ’C’ language. The fuzzy automata modelling approximate string matching is proposed for error handling during lexical analysis. By introducing fuzziness to lexemes the typographical errors can be rectified at the time of compilation and flexibility of lexical analyser can be greatly improved. The recognition of fuzzy tokens during lexical analysis is described in order to correct errors caused by sticking key, deletion, typing and swapping key in keywords during C programming. Algorithms and pseudo code are being developed to measure the degree of membership of crisp and fuzzy lexemes. Accuracy is tested and examined once the fuzzy lexemes are trained using a neural network. The proposed method is an add on feature that can be incorporated in existing compilers and IDEs to increase their flexibility.

Keyword: Fuzzy lexemes, fuzzy automata, error handling, approximate string matching, fuzzy lexical analysis

1. Introduction

Approximate string matching (ASM) is used in many applications such as information retrieval, text searching, text summarization, pattern recognition, spell checking, recognition of lexemes in lexical analyser etc. [7, 11–14, 18, 19, 21, 30]. ASM is a technique to compute similarity between a pair of strings. In literature ASM is broadly classified into string based similarity measures and term based similarity measures [11, 30]. There exists many approaches to compute similarity in ASM such as distance metrics [11, 22], n-gram methods [2], automata based methods [9, 23] and filtering algorithms [6]. The distance measure based on Damerau-Levenshtein edit distance [15, 22] counts minimum number of operations required to transform one string into another, including operations such as insertion, deletion, substitution, swapping of symbols having time complexity $O(mn)$. Extensions of distance method proposed in [31] with k difference allowed has time complexity $O(kn)$ and when alphabet size b is taken into consideration complexity is $O(kn/\sqrt{b} - 1)$. Mendivil et al. [17] applied fuzzy automata to compute similarity between two strings by considering the membership value of fuzzy language assigning cost to different edit operations. Garitagoitia et al. [18] introduced fuzzy automata to compute similarity between strings by considering fuzzy transitions based on assigning predefined cost to each edit operation. Astrain et al. [13] created a dictionary of all the possible errors and then computed similarity of input string by comparing it with every word of dictionary. They proposed fuzzy automaton for classification of strings and attained improved recognition rates using generic algorithm with time complexity $O(m \times n \times t)$, where t is size

*Corresponding author. Ranjeet Kaur, E-mail: reetmaths@gmail.com.

of alphabet and m, n are lengths of target and input strings, respectively.

Astrain et al. [14] proposed fuzzy automata with ϵ moves that allows fuzzy edit operations modelled by binary fuzzy relations. They suggested fuzzy measure to compute similarity using string alignment and edit operations to overcome limited number of errors. Authors [28] suggested a technique to search or match strings in special cases when some pairs of symbols are more similar to each other and discussed its applications in DNA computing and spellchecker.

Ravi et al. [21] introduced intuitionistic fuzzy automata for ASM to compute similarity as well as dissimilarity of all possible edit operations. Ko et al. [25] computed similarity using context free grammar and fuzzy automata considering linear, affin and concave gap costs using dynamic programming algorithms. Shaprio et al. [7] initiated deep learning model in combination with memoization for text classification to detect and correct spelling errors in OCR output. Boinski et al. [26] suggested n-grams methods for spelling errors and correction.

Essex [1] came up with secure approximate string matching for record linkage against errors using bigram decomposition and extended dice coefficient to threshold dice coefficient for string similarity. Recently, Faro et al. [24] introduced Range automaton by proposing Backward Range Automata Matcher algorithm and added features of swap matching and multiple string matching for text searching algorithms with quadratic run time complexity.

1.1. Research gap

In literature, there are edit distance, hamming distance, Levenshetein, Damerau-Levenshetein, n-grams, q-grams methods and algorithms developed so far for approximate string matching [11, 30]. The application of ASM in lexical analysis is not explored much. Crisp compilers use a bi-valued approach. In compilers, a string is either recognised or not recognised as a token. There is no technique for compilers in the 'C' language to accommodate typographical errors in existence. Mateescu et al. [3] discussed the applications of ASM and fuzzy automata in lexical analysis and proposed an extension of LEX compiler for UNIX with added features to deal with typographical errors. Bhosle et al. [27] has suggested a fuzzy lexical analyser for tiny language and proposed algorithm to consider typographical error during token recognition process. Their method is based on threshold value for acceptance of error keywords but it is limited to only eight keywords and ignores swapping key errors.

1.2. Basis of Study

The primary goal of our research is to use fuzzy automata in lexical analysis phase of compiler construction to allow lexical errors in keywords which may exist due to typographical errors taking edit operations of ASM into consideration. The typographical error or typo error refers to the unintentional typing error that occurs due to accidental hitting a wrong key. There are four kinds of typographical errors: deletion of key, swapping key, sticking key and typing key error. The motive of the research is to handle typographical errors in keywords by compiler itself instead of user intervention.

Existing IDEs deal with this issue in two ways:
Case1: It recommends the feasible keywords while typing.
Case 2: After compiling, they display a message i.e. not defined or implicit declaration of function.
In the first case, programmer must opt correct suggestion and in case 2, one has to rectify these errors at the place manually. Whereas by proposed method, these errors can be handled at the time of compilation. Thus, it will save energy and time of the programmer. The proposed idea is an add on feature that can be incorporated in the existing compilers, where instead of showing errors it will accept keywords, on the basis of their membership values.

1.3. Contribution

String matching is used to implement lexical analyser which requires regular expressions and finite automata. Fuzziness in lexemes for token keyword is introduced to boost the flexibility of crisp compilers and to make them more user friendly. In proposed work, the keywords are categorised into crisp keywords and error keywords. Both are fuzzy in nature. Crisp keywords are correctly spelled keywords directly accepted by compilers and probable typing errors that result from sticking key (deletion), missing letter (insertion), sequencing (swapping), or mistyping of keys (substitution) are termed as error keywords or fuzzy keywords. Fuzzy regular expressions of keywords are generated in fuzzy lexical analysis. Fuzzy regular expressions are first converted into Non-deterministic Finite Automata with Fuzzy (final) States (FS-NFA) and then Deterministic Finite Automata with Fuzzy (final) States(FS-

DFA) are constructed because FS-DFA are easy to simulate.

Algorithms along with pseudo code for calculating the membership value of crisp keywords and error keywords are proposed. At various levels of membership, the error keywords are accepted. These membership values are trained using a neural network, which enables lexical analyser to consider keywords with different membership value.

The present work is an effort to discuss ASM for handling typographical errors in tokens generated at the time of lexical analysis phase of C compiler. In the proposed approach, all the possible error keywords are generated corresponding to crisp keyword, along with edit operation we have added method to calculate the similarity between crisp and error keyword by considering membership value of generated error keywords.

The paper is laid out in following manner: The first section is an introduction. In section 2, some important concepts are recalled. Section 3, describes proposed method and procedure. The results are covered in Section 4, and the study concludes in Section 5.

2. Preliminaries

This section goes over some basic definitions of fuzzy languages and fuzzy automata. For more information, readers can refer [3, 8] and [29]. Throughout this paper Σ denotes any set of finite alphabets and Σ^* be the set of finite strings constructed from elements of Σ.

Definition 2.1. [8] A fuzzy language is collection of all strings having membership values in the interval [0, 1].

Definition 2.2. [3, 8] Let $f_{\tilde{\mathcal{L}}_1}$ an $f_{\tilde{\mathcal{L}}_2}$ be two fuzzy languages over Σ, Then the operations on $f_{\tilde{\mathcal{L}}_1}$ and $f_{\tilde{\mathcal{L}}_2}$ are defined as:

1. $f_{\tilde{\mathcal{L}}} = f_{\tilde{\mathcal{L}}_1} \cup f_{\tilde{\mathcal{L}}_2}$ is union of fuzzy languages $f_{\tilde{\mathcal{L}}_1}$ and $f_{\tilde{\mathcal{L}}_2}$ and its membership function is $\mu_{f_{\tilde{\mathcal{L}}}}(x) = max\{\mu_{f_{\tilde{\mathcal{L}}_1}}(x), \mu_{f_{\tilde{\mathcal{L}}_2}}(x)\},\ x \in \Sigma^*$.
2. $f_{\tilde{\mathcal{L}}} = f_{\tilde{\mathcal{L}}_1} \cap f_{\tilde{\mathcal{L}}_2}$ is intersection of fuzzy languages $f_{\tilde{\mathcal{L}}_1}$ and $f_{\tilde{\mathcal{L}}_2}$ and its membership function is $\mu_{f_{\tilde{\mathcal{L}}}}(x) = min\{\mu_{f_{\tilde{\mathcal{L}}_1}}(x), \mu_{f_{\tilde{\mathcal{L}}_2}}(x)\},\ x \in \Sigma^*$.
3. $f_{\tilde{\mathcal{L}}} = {f_{\tilde{\mathcal{L}}_1}}^c$ is complement of fuzzy language $f_{\tilde{\mathcal{L}}_1}$ and its membership function is $\mu_{f_{\tilde{\mathcal{L}}}}(x) = 1 - \mu_{f_{\tilde{\mathcal{L}}_1}}(x),\ x \in \Sigma^*$.
4. $f_{\tilde{\mathcal{L}}} = f_{\tilde{\mathcal{L}}_1}.f_{\tilde{\mathcal{L}}_2}$ is concatenation of fuzzy languages $f_{\tilde{\mathcal{L}}_1}$ and $f_{\tilde{\mathcal{L}}_2}$ and its membership function is $\mu_{f_{\tilde{\mathcal{L}}}}(x) = max\{min(\mu_{f_{\tilde{\mathcal{L}}_1}}(a), \mu_{f_{\tilde{\mathcal{L}}_2}}(b)) \mid x = ab,\ a, b \in \Sigma^*\},\ x \in \Sigma^*$.
5. $f_{\tilde{\mathcal{L}}} = {f_{\tilde{\mathcal{L}}_1}}^*$ is Kleene closure or star operation of $f_{\tilde{\mathcal{L}}_1}$ and its membership function is $\mu_{f_{\tilde{\mathcal{L}}}}(x) = max\{min(\mu_{f_{\tilde{\mathcal{L}}_1}}(x_1), ..., \mu_{f_{\tilde{\mathcal{L}}_1}}(x_n)) \mid x = x_1, x_2...x_n,\ x_1, x_2, ..., x_n \in \Sigma^*,\ n \geq 0\}$, $x \in \Sigma^*$, assuming that $min\ \phi = 1$.
6. $f_{\tilde{\mathcal{L}}} = {f_{\tilde{\mathcal{L}}_1}}^+$ is Kleene plus operation of $f_{\tilde{\mathcal{L}}_1}$ and its membership function is $\mu_{f_{\tilde{\mathcal{L}}}}(x) = max\{min(\mu_{f_{\tilde{\mathcal{L}}_1}}(x_1), ..., \mu_{f_{\tilde{\mathcal{L}}_1}}(x_n)) \mid x = x_1x_2...x_n,\ x_1, x_2, ..., x_n \in \Sigma^*,\ n \geq 1\},\ x \in \Sigma^*$.

Remark 1. The equivalence and inclusion relations are true in fuzzy languages.

Definition 2.3. [3] Let $f_{\tilde{\mathcal{L}}}$ be a fuzzy language over Σ and $\mu_{f_{\tilde{\mathcal{L}}}} : \Sigma^* \to [0, 1]$ be the membership function of $f_{\tilde{\mathcal{L}}}$, then for each $t \in [0, 1]$, the set $S_{f_{\tilde{\mathcal{L}}}}(t)$ is defined as $S_{f_{\tilde{\mathcal{L}}}}(t) = \{x \in \Sigma \mid \mu_{f_{\tilde{\mathcal{L}}}}(x) = t\}$. $f_{\tilde{\mathcal{L}}}$ is called regular fuzzy language if

1. the set $\{t \in [0, 1] \mid S_{f_{\tilde{\mathcal{L}}}}(t) \neq \phi\}$ is finite.
2. for each $t \in [0, 1]$, $S_{f_{\tilde{\mathcal{L}}}}(t)$ is regular.

Example 2.1. Let $f_{\tilde{\mathcal{L}}}$ be a language over $\Sigma = \{0, 1\}$ where $\mu_{f_{\tilde{\mathcal{L}}}}$ is defined as

$$\mu_{f_{\tilde{\mathcal{L}}}}(x) = \begin{cases} 1, & if\ x \in 01^* \\ 0.76, & if\ x \in 011^*0 \\ 0.59, & if\ x \in 1011^* \\ 0, & otherwise. \end{cases}$$

Then $f_{\tilde{\mathcal{L}}}$ is a regular fuzzy language , since $t = \{0, 0.59, 0.76, 1\}$ is finite and for each $t \in [0, 1]$, $S_{f_{\tilde{L}}}(t)$ is a regular language.

Regular expressions are used to represent strings in more declarative way in the form of algebraic expressions [4].

Definition 2.4. [3] A modified regular expression is termed as Fuzzy Regular Expression(FRE).

1. $(\nabla)/t$ is a FRE where ∇ is regular expression over Σ and $t \in [0, 1]$.
2. The union $\nabla_1 + \nabla_2$, concatenation $(\nabla_1).(\nabla_2)$ and Kleene closure $(\nabla_1)^*$ of fuzzy regular expressions are all fuzzy regular expressions.

3. FRE is obtained by repeated applications of (i) and (ii) finite times.

Definition 2.5. FS-NFA [3] FS-NFA is a 5-tuple $M = \{S, \Sigma, \delta, s_0, F_M\}$ where:

1. S is a finite non-empty set of states.
2. Σ is a finite non-empty set of input symbols.
3. $\delta : S \times \Sigma \times S \to [0, 1]$, is the transition function.
4. $s_0 \in S$ is called initial state.
5. $F_M : S \to [0, 1]$ is the degree function for fuzzy final-state set.

For $x \in \Sigma^*$ and $s_1, s_2 \in S$, define $\delta^* : S \times \Sigma^* \times S \to [0, 1]$

$$\delta^*(s_1, \lambda, s_2) = \begin{cases} 0, \ if \ s_1 \neq s_2 \\ 1, \ if \ s_1 = s_2 \end{cases}$$

and $\delta^*(s_1, x, s_2) = \vee\{\delta^*(s_1, x', s) \wedge \delta(s, a, s_2) | s \in S\} \ \forall \ x = x'a, \ x' \in \Sigma^*, \ a \in \Sigma.$

A string $x \in \Sigma^*$ is accepted by FS-NFA with degree or membership value $d_M(x)$, where $d_M(x) = max\{F_M(s) \mid (s_0, x, s) \in \delta^*\}$.
The fuzzy language accepted by FS-NFA is denoted as $L(M) = \{(x, d_M(x)) | x \in \Sigma^*\}$.

Remark 2. FS-DFA is an FS-NFA [3] with only difference in transition function i.e. $\delta : S \times \Sigma \to S$.

A string $x \in \Sigma^*$ is accepted by FS-DFA with degree $d_M(x)$, where $d_M(x) = F_M(s) \ni s = \delta^*(s_0, x)$ and $d_M(x) = 0$ if $\delta^*(s_0, x)$ is not defined.

3. Methodology

Compilers convert a source language to a target language and report any errors back to the source language. Analysis and synthesis are the two phases of the compilation process. The first step in the analysis phase is lexical analysis, which involves breaking down the initial source language into parts and generating an intermediate representation. It is the interface between source language and compilers. During lexical analysis the source language is grouped into lexemes, which are meaningful sequence of words. For parsing, the lexemes are subsequently transformed to tokens. Tokens can be treated as identifiers, keywords, constants, operators and punctuation symbols. Every lexeme that is to be identified as a token follows a set of predefined grammar rules. Consider program code in C language:

$$float\ product\ (float\ num1,\ \ float\ num2)$$
$$\{$$
$$//This\ will\ multiply\ 2\ numbers$$
$$return\ (num1\ *\ num2);$$
$$\}$$

Table 1
Tokens and **Lexemes**

Lexeme	Token
float	keyword
product	identifier
(	punctuation symbol
num1	identifier
,	punctuation symbol
num2	identifier
)	punctuation symbol
{	punctuation symbol
return	keyword
*	operator
;	punctuation symbol
}	punctuation symbol

The tokens and lexemes for this program code is listed in Table 1.

Compilers in 'C' language do not support typographical errors in keywords., therefore Fuzzy Regular Expressions (FRE) for all 32 keywords [5] are generated, taking into account all conceivable errors. The algorithm 1 and algorithm 3 are suggested to calculate the membership values of crisp keywords and error keywords respectively. For all keywords, FS-NFA is constructed and for minimization FS-NFA is converted to FS-DFA.

Table 2 lists some possible typographical keyword errors.

– Sticking key: particular character is typed twice or thrice
– Deletion key: particular character is not typed
– Swapping key: sequence of character is mistyped
– Typing key: typing of adjacent key in QWERTY keyboard

Fuzzy Regular Expression: For each keyword, fuzzy regular expressions are generated that allow for sticking, deletion, swapping, and typing errors.

The FRE for reserved word "*float*" is as follows:

Table 2
Possible typographical errors

Keyword	Sticking Key Error	Deletion Error	Swapping Key Error	Typing Error
float	fffloaaat	floa	ftola	dloat
while	whiiileee	hile	wlihe	wgile
switch	switcchh	swtch	wsithc	sqitch
printf	prriiintt	print	prnift	ptimtd
void	voiiiid	vid	viod	boif
struct	ssttructt	stuct	srtcut	ateuct

$float/1 + (ff^{+}ll^{*}oo^{*}aa^{*}tt^{*} + ff^{*}ll^{+}oo^{*}aa^{*}tt^{*} + ff^{*}ll^{*}oo^{+}aa^{*}tt^{*} + ff^{*}ll^{*}oo^{*}aa^{+}tt^{*} + ff^{*}ll^{*}oo^{*}aa^{*}tt^{+})/m_1 + (floa + flot + flat + loat + foat)/m_2 + (fotla + flaot + lfoat + lfaot + oflat + olfta + afolt + aotfl + tfloa + tofla)/m_3 + ((r + d + g + c)loat + f(o + k + p + i)oat + fl(i + l + p)at + flo(s + z + q)t + floa(r + y + g))/m_4$
Here, m_1 denotes membership value due to sticking key error, m_2 denotes membership value due to deletion key error, m_3 denotes membership value due to swapping key error and m_4 denotes membership value due to typing key error.

Algorithm 1 and its pseudo code are proposed to calculate the membership values of crisp keywords. The degree of letter specified in algorithm represents the membership value of letters placed at different position or index of crisp keyword contributing to calculate the membership value of crisp keyword. Each letter is contributing in formation of crisp keyword therefore the value of each letter is assumed to be same. Thus, the membership value of crisp keyword is calculated by determining the degree of each letter present in the keyword.

Algorithm 1. Algorithm to find membership value of 32 crisp keywords for 'C' Language

Step-1: Create a dictionary for all 32 keywords

Step-2: Find length of each keyword, $length(crisp\_keyword)$.
Step-3: Initialise membership value of each keyword as 0, $membership\_value(crisp\_keyword) = 0.0$.
Step-4: Find degree of each letter in crisp keyword,

$degree(letter) = 1/length(crisp\_keyword)$.
Step-5: Calculate membership value of each keyword using for loop
$0 \leq i < length(crisp\_keyword)$,
$membership\_value(crisp\_keyword) = membership\_value(crisp\_keyword) + degree(letter)$.

Initially the algorithm 1 has time complexity as $O(m * n)$, where m is the length of keyword and n is the number of keywords i.e. 32. Since all keywords are of small length, m can be assumed to be constant. Hence, time complexity of algorithm 1is $O(n)$.

Algorithm 2. Pseudo code
```
Input  : 32 keywords in keywordArray[32]
            // a string array of length 32
to store all crisp keywords.
Output: membershipValue(crispKeyword)
Initialise: lengthArray[32] ←{0}
          // to store length of each crisp
keyword.
Initialise: j ← 0
    // to iterate over lengthArray[32]
1 for i ← 0 to lengthArray[32] do
2     lengthArray[j] ← length(keywordArray[i])
3     j ← j+1
  Initialise: membershipValue[32] ← {0}
4 for i ← 0 to length(keywordArray[32]) do
      Initialise: lettersDegreeArray[26] ← {0}
5     letter=sort(keywordArray[i])
6     degree(letter)← 1/length(keywordArray[i])
7     Store degree(letter) accordingly in lettersDegreeArray[26]
8     for l ← 0 to length(keywordArray[i] do
9         membershipValue[i]← membershipValue[i]+degree(letter)
      Output: membershipValue[i]
```

Algorithm 1 is explained with the help of example 3.1.

Example 3.1. To find membership value of keyword "struct"
$crisp\_keyword =$ "struct"
$length(crisp\_keyword) = length(struct) = 6$
Assume, $membership\_value(struct) = 0.0$

Now calculate degree of each letter present in "struct" using, $degree(letter) = 1/length(crisp\_keyword)$

$degree(s) = \frac{1}{6},\ \ degree(t) = \frac{1}{6},\ \ degree(r) = \frac{1}{6}$

$degree(u) = \frac{1}{6},\ \ degree(c) = \frac{1}{6},\ \ degree(t) = \frac{1}{6}$

$membership\_value(crisp\_keyword) += degree(letter)$

$membership\_value(struct) = 0.0 + \frac{1}{6}$

$membership\_value(struct) = \frac{1}{6} + \frac{1}{6}$

$membership\_value(struct) = \frac{2}{6} + \frac{1}{6}$

$membership\_value(struct) = \frac{3}{6} + \frac{1}{6}$

$membership\_value(struct) = \frac{4}{6} + \frac{1}{6}$

$membership\_value(struct) = \frac{5}{6} + \frac{1}{6}$

$membership\_value(struct) = 1.0$

All the possible errors have been considered, i.e. sticking key, deletion, swapping key and typing errors to find error keywords corresponding to each crisp keyword. Algorithm 3 is proposed for all these errors to calculate membership value of error keywords. The index of the letter plays an important role in proposed algorithm to calculate membership values of keywords. The membership value of error keywords is calculated by considering four different possible cases:

1. In deletion of key error the length of error keyword is less than crisp keyword, hence the $degree(letter) = 0$ for missing letter, rest all $degree(letter)$ will be same as in the crisp keyword.
2. In QWERTY typing key error the degree of letter which is not matched at index will be considered as $degree(letter) = 0$, rest all $degree(letter)$ will be same as in the crisp keyword.
3. In sticking key error the length of error keyword will be greater than the crisp keyword, hence for sticking letter occurrence of letter is taken into account.
4. In swapping key error, the degree of letter in error keyword is depending on the $degree(letter)$ of crisp keyword and count of interchanges, i.e. how many positions are required to shift the swapped letter.

Algorithm 3. Algorithm to find membership value of *error_keyword*.

Input: $crisp\_keyword$**,** $error\_keyword$
Output: $membership\_value(error\_keyword)$

Step-1 Initialise
$membership\_value(error\_keyword) = 0.0$

Step-2 Calculate $length(error\_keyword)$ and $length(crisp\_keyword)$.

Step-3 Three cases may arise:

Step-3.1 Case-I
If lengths of both keywords are equal then it will be considered as swapping key error or typographical error.

Step-3.1.1 If sorting list of letters of *error_keyword* and *crisp_keyword* is equal then it is considered as swapping key error otherwise typographical error.

Step-3.1.1.1 Swapping key error: The *index* is searched where mismatch occurs in *crisp_keyword* and *error_keyword*. Count the number of interchanges [CI] required to shift the mismatched letter to its original position.

Step-3.1.1.2 To restrict the membership value of *error_keyword* in interval [0,1], α/CI, where $\alpha \in [0, 1]$ is considered whenever swapping key error is observed. Calculate
$degree(letter) = \alpha/CI * length(crisp\_keyword)$
$membership\_value(error\_keyword) += degree(letter)$

Step-3.1.2 Typographical error: Typing error in QWERTY keyboard by possible adjacent keys. Both keywords are compared at each index,
if $error\_keyword[index] \neq crisp\_keyword[index]$, then $degree(letter) = 0$
else $degree(letter) = 1/length(crisp\_keyword)$
$membership\_value(error\_keyword) += degree(letter)$

Step-3.2 Case-II

If length of error keyword is greater than length of crisp keyword it will be considered as sticking key

error. Count the number of times each letter appears in $error\_keyword$. i,e. $occ(letter)$
$degree(error\_keyword[index]) = 1/occ(letter) * length(crisp\_keyword)$.
$membership\_value(error\_keyword) += degree(letter)$

Step-3.3 Case-III

If length of error keyword is less than length of crisp keyword it will be considered as deletion key error.
$membership\_value(error\_keyword) = length(error\_keyword)/length(crisp\_keyword)$.

Step-4 Write membership value of $error\_keyword$ calculated in Step 3.

Step-5 End.

The time complexity of algorithm 3 is $O(mlogm)$, where m is the length of error keyword.

Algorithm 3 is explained with the help of example 3.3. Python code is created to generate these error keywords and to compute membership value of error keywords. The greater the membership value of error keyword, the more it is considered similar to the crisp keyword.

Example 3.2. Consider the crisp keyword "float" and error keyword "ffloaat".

Step-1
$crisp\_keyword =$"float"
$error\_keyword =$ "ffloaat"

Step-2 Initialise
$membership\_value$("ffloaat")= 0.0

Step-3
$length(error\_keyword) = length$("ffloaat")= 7
$length(crisp\_keyword) = length$("float")= 5

Step-4 Here,
$length$("ffloaat")> $length$("float"), therefore case II sticking key error of algorithm 3 arises.

Step-4.2 Find occurrences of each letter in "ffloaat",

$occ(f) = 2,\ occ(l) = 1,\ occ(o) = 1,\ occ(a) = 2,\ occ(t) = 1$
$degree(error\_keyword[0]) = degree(f) = \frac{1}{2*5} = \frac{1}{10}$
$degree(l) = \frac{1}{1*5} = \frac{1}{5}$

$degree(o) = \frac{1}{1*5} = \frac{1}{5}$

$degree(a) = \frac{1}{2*5} = \frac{1}{10}$

$degree(t) = \frac{1}{1*5} = \frac{1}{5}$

$membership\_value(error\_keyword) += degree(letter)$

Step-5 $membership\_value$("ffloaat")= 0.8

Table 3 represents summary of some possible typographical error keywords corresponding to crisp keyword *"float"* and their membership values are computed using algorithm 3.

Every language that can be accepted by FS-NFA can also be accepted by FS-DFA [16]. FS-NFA for keyword *"float"* is given in Fig. 1 containing 52 states. Corresponding to FS-NFA using subset construction, method FS-DFA is constructed. FS-DFA for keyword *"float"* is given in Fig. 2 which has less number of states as compared to FS-NFA.

3.1. Implementation

This section discusses the suggested method’s general implementation.

Step-I First all the keywords of ‘C’ language are defined.

Step-II All the possible errors of each keyword are generated using fuzzy regular expressions and dictionary is created which consists of crisp keywords and error keywords.

Step-III Membership value of each error keyword is calculated using proposed algorithm 1 and algorithm 3 considering all errors.

Step-IV Each keyword is assigned a label.

Step-V A neural network is constructed to train error keywords corresponding to these labels.

Step-VI Accuracy is measured on testing data.

Table 3
Membership Values of error keywords generated from crisp keyword *"float"*

error keyword	deg(f)	deg(l)	deg(o)	deg(a)	deg(t)	member-ship value
fflooat	0.1	0.1	0.1	0.2	0.2	0.7
floatt	0.2	0.2	0.2	0.2	0.1	0.9
floa	0.2	0.2	0.2	0.2	0	0.8
dloat	0	0.2	0.2	0.2	0.2	0.8
ftola	0.2	0.04	0.2	0.04	0.03	0.51
folat	0.2	0.12	0.03	0.2	0.2	0.75

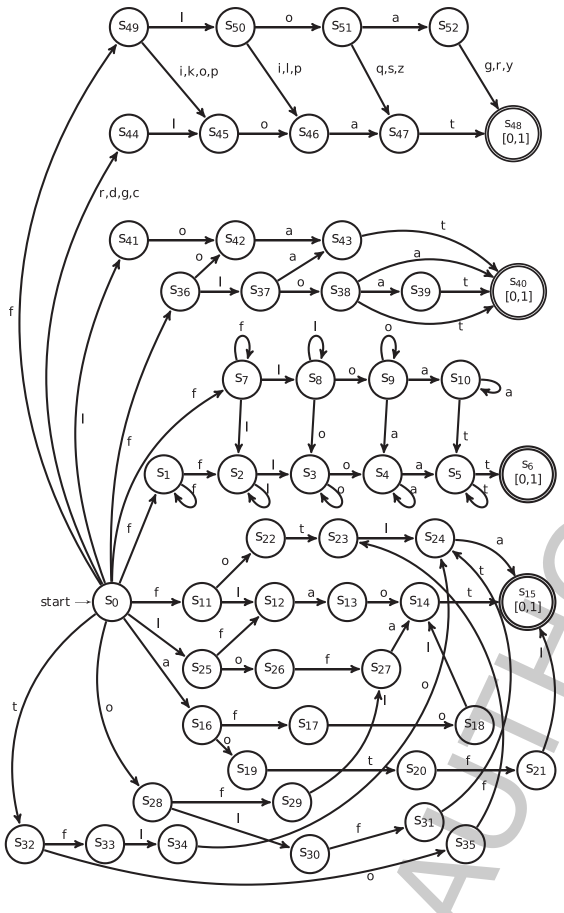
Fig. 1. FS-NFA for keyword "float".

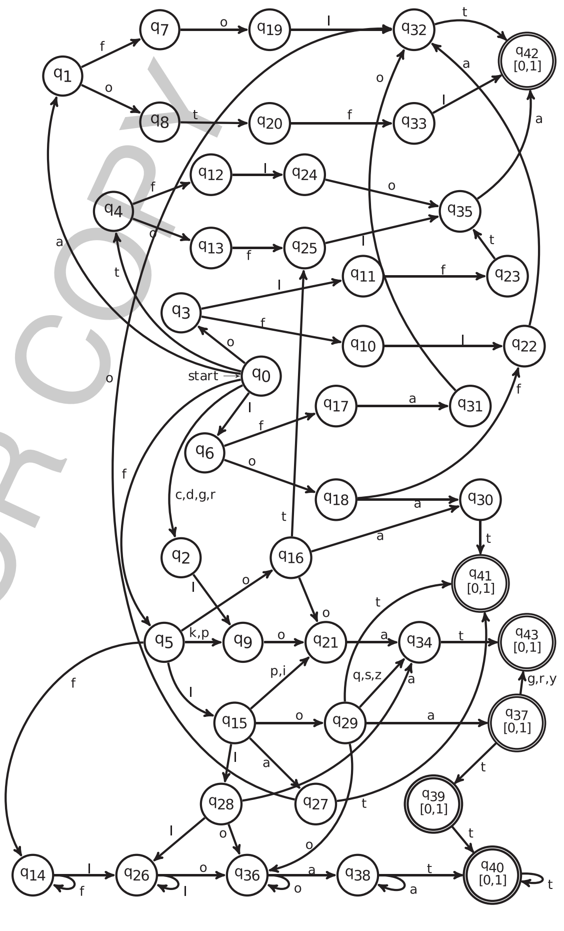
Fig. 2. FS-DFA for keyword "float".

4. Results

For experimentation, a dictionary is created for all the 32 crisp keywords for 'C' programming language using python. Corresponding to each crisp keyword, 40 keywords are generated by considering all the possible typographical errors. The dictionary contains 1280 keywords after appending all the errors considering deletion, sequencing, sticking key and typing errors. Each keyword is assigned a label and added with crisp keyword corresponding to that label. By reducing redundant comparison, accuracy is increased, and time complexity is decreased. The membership value of each keyword is calculated using the algorithm 1 and algorithm 3. The efficiency of method is further enhanced by using neural network to validate it for unseen data.

The data is split for training and testing using Keras framework. For training set 80 percent i.e. 1024 keywords are taken in to consideration. A neural network is trained for these membership values. The network is run with learning rate 0.7 and 90 epochs. The average accuracy measured is 82.81%. The loss graph in

Table 4
Comparative Analysis of membership values of error keywords

	Ravi et al.[21]	Vaishali et al.[27]	Kondark [10]	Proposed Algorithm
ffloaat	0.6	0.8	0.8	0.8
ftola	0.6	1	0	0.51
folat	0.6	1	0.25	0.75
floa	0.6	08	0.86	0.8
dloat	0.6	0.8	0.75	0.8

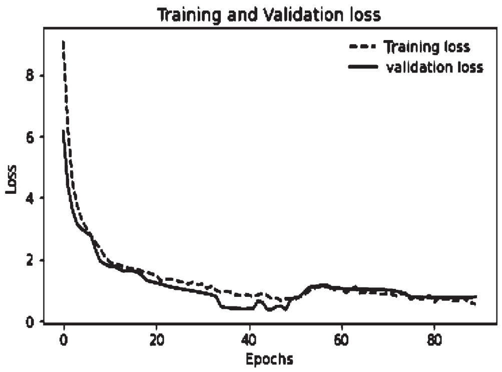

Fig. 3. Training and Validation Loss.

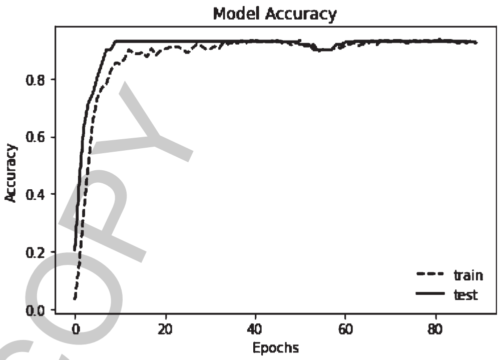

Fig. 4. Model Accuracy.

Fig. 3 depicts that, for the first 12 epochs, the loss in the training set falls off quickly. The loss for the test set does not decline as quickly as it does for the training set, but rather, after 40 epochs, it nearly remains constant. This shows that the model generalizes well to new data. Similarly, Figure 4 demonstrates how quickly accuracy rises in the first 10 epochs for both training and testing sets, hence the network is learning fast. Afterwards, the curve flattens indicating that the fewer training epochs are needed to train the model further.

4.1. Comparative Analysis

The comparative analysis of calculating the membership values of keywords using suggested algorithms from the literature is presented in Table 4. In Ravi et al. [21], as the cost of all possible edit operations are predefined, therefore the membership values of error keywords are constant. In [27], the swapping of keys is not considered. In [10], while switching adjacent keys yields accurate results, switching all the letters in a word is not taken into consideration. The results demonstrate that the membership values obtained by our proposed algorithm are near to expected membership value as compared to other three methods in terms of recognition, encompassing all feasible scenarios.

4.2. Recommendations

In the proposed work, application of ASM using fuzzy automata is discussed in lexical analysis phase of C- compiler. The proposed idea is an add on feature which can be incorporated in existing compilers not only in ’C’ language but also in other compilers and IDEs. The authors team is not boasting to develop a new compiler, but different OS companies or programmers can take this approach as a new bench mark for future development of an efficient and flexible product.

5. Conclusion

The proposed method is a feature that can enhance the efficiency of existing compilers and make compilers more user friendly. The errors are rectified at the time of lexical analysis phase. This saves time for parsing and further analysis of source program. Traditional compiler of ‘C’ language does not support error handling in keywords, hence to improve the efficiency of existing compilers fuzziness in the token is introduced. Considering operations of ASM possible error keywords are compared with crisp keywords. Algorithms are proposed so that error keywords may be accepted at varying degrees of their membership values.

The keywords are further trained using neural network and an average classification accuracy of 83% is achieved.

In future, the method can be extended for operators and other tokens and full implementation of a fuzzy lexical analyser can be done. The proposed approach can be further tested and implemented in other compilers, IDEs to increase their flexibility and user friendliness.

References

[1] A. Essex, Secure Approximate String Matching for Privacy - preserving Record Linkage, *IEEE Transactions on Information Forensics and Security* **14**(10) (2019), 2623–2632.

[2] A.M. Robertson and P. Willett, Applications of n-grams in Textual Information Systems, *Journal of Documentation* **54**(1) (1998), 48–67.

[3] A. Mateescu, A. Salomaa, K. Salomaa and S. Yu, Lexical Analysis with a Simple Finite Fuzzy Automaton Model, *Journal of Universal Computer Science* **1**(5) (1995), 292–311.

[4] A.V. Aho, R. Sethi and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley series in computer science, 1986.

[5] B.W. Kernighan and D.M. Ritchie, The C Programming Language Prentice Hall, second edition, 1988.

[6] C. Li, J. Lu and Y. Lu, Efficient Merging and Filtering Algorithms for Approximate String Searches, *IEEE 24th International Conference on Data Engineering* (2008), 257–266.

[7] D. Shapiro, N. Japkowicz, M. Lemay and M. Bolic, Fuzzy String Matching with a Deep Neural Network, *Applied Artificial Intelligence* **32**(1) (2018), 1–12.

[8] E.S. Santos, Maximin automata, *Information and Control* **13**(4) (1968), 363–377.

[9] E. Ukkonen and D. Wood, Approximate String Matching with Suffix Automata Algorithmica **10** (1993), 353–364.

[10] G. Kondrak, N-Gram Similarity and Distance. In: Consens, M., Navarro, G. (eds) String Processing and Information Retrieval, *SPIRE 2005. Lecture Notes in Computer Science* 3772. Springer, Berlin, Heidelberg (2005)

[11] G. Navarro, A Guided Tour to Approximate String Matching, *ACM Computing Surveys* **33**(1) (2001), 31–88.

[12] G. Navarro and K. Fredriksson, Average Complexity of Exact and Approximate Multiple String Matching, *Theoretical Computer Science* **321** (2-3) (2004), 283–290.

[13] J.J. Astrain, J.R. Garitagoitia, J.R.G. de Mendivil, J. Villadangos and F. Farina, Approximate String Matching Using Deformed Fuzzy Automata: A Learning Experience, *Fuzzy Optimization and Decision Making* **3** (2004), 141–155.

[14] J.J. Astrain, J.R.G. de Mendivil and J.R. Garitagoitia, Fuzzy Automata with ϵ-moves Compute Fuzzy Measures between Strings, *Fuzzy Sets and Systems* **157**(11) (2006), 1550–1559.

[15] J.L. Peterson, Computer programs for detecting and correcting spelling errors, *Communications of the ACM* **23**(12) (1980), 676–687.

[16] J.N. Mordeson and D.S. Malik, Fuzzy Automata and Languages Theory and Applications, *CRC Press*, 2002.

[17] J.R.G. de Mendivil, J.R. Garitagoitia, J.J. Astrain and J. Echanobe, Fuzzy automata for imperfect string matching, *Proceedings of ESTYLF* (2000), 527–532.

[18] J.R. Garitagoitia, J.R.G. de Mendivil, et al., Deformed Fuzzy Automata for Correcting Imperfect Strings of Fuzzy Symbols, *IEEE Transactions on Fuzzy Systems* **11**(3) (2003), 299–310.

[19] J. Taeho, Automatic Text Summarization Using String Vector Based K Nearest Neighbor, *Journal of Intelligent & Fuzzy Systems* **35**(6) (2018), 6005–6016.

[20] Kenneth C. Louden, Compiler Construction Principles and Practice, *PWS Publishing Company*, 1997.

[21] K.M. Ravi, A. Choubey and K.K. Tripathi, Intuitionistic Fuzzy Automaton for Approximate String Matching, *Fuzzy Information and Engineering* **6**(1) (2014), 29–39.

[22] P.A.V. Hall and G.R. Dowling, Approximate String Matching, *Computing Surveys* **12** (1980), 381–402.

[23] R. Baeza-Yates and G. Navarro, Faster Approximate String Matching, *Algorithmica*, **23**(2) (1999), 127–158.

[24] S. Faro and S. Scafiti, A weak approach to suffix automata simulation for exact and approximate string matching, *Theoretical Computer Science* **933** (2022), 88–103.

[25] S.K. Ko, Y.S. Han and K. Salomaa, Approximate matching between a context-free grammar and a finite-state automaton, *Information and Computation* **247** (2016), 278–289.

[26] T. Boinski, A. Zimnicki, et al., Evaluating Asymmetric Ngrams as Spell-Checking Mechanism, *11th International Conference on Human System Interaction* (2018), 356–361.

[27] V. Bhosle and S.R. Chaudhari, Fuzzy Lexical Analyser: Design and Implementation, *International Journal of Computer Applications* **123**(11) (2015), 1–7.

[28] V. Snasel, A. Abraham, A. Keprt and A.E. Hassanien, Approximate String Matching by Fuzzy Automata, *In Man-Machine Interactions* (2009), pp. 281–290.

[29] W.G. Wee and K.S. Fu, A formulation of fuzzy automata and its application as a model of learning systems, *IEEE transactions on System science and Cybernetics* **5**(3) (1969), 215–223.

[30] W.H. Gomaa and A.A. Fahmy, A Survey of Text Similarity Approaches, *International Journal of Computer Applications* **68**(13) (2013), 13–18.

[31] W.I. Chang and J. Lampe, Theoretical and empirical comparisons of approximate string matching algorithms, *Combinatorial Pattern Matching* (1992), pp. 175–184.