T9 Predictive Text Technology for Telephone Keypads Using Regular Expressions

Alternative T9 prediction method to the conventional trie data structure usage

James Chandra 13519078
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: tweenlicious@gmail.com

***Abstract*—In an age of rapid technological development, and the instantification of many different aspects of life, retrospectives ought to be done to enhance general understanding of the history of technological development, and even find new ways to innovate current technologies by looking at ones deemed obsolete. One of these examples are the obsolescence of T9 predictive text keyboards built for twelve-key telephone or mobile phone mechanical keypads that were replaced by the dawn of capacitive on-touchscreen keyboards at the early turn of the twenty-first century. The T9 had predictive text capabilities which allowed for users to spell words at a faster pace, utilizing the trie data structure as a dictionary to store pre-determined word predictions. This paper will explore alternative methods to the trie usage, namely using regular expressions—which was not deemed viable or feasible in the past due to the limited computing power that mobile cell phones had at the time, and considering how expensive regular expression operations can be, having the potential to even exhibit exponential behavior given the appropriate strings or circumstances—to generate word predictions by a converse comparison method to the conventional method.**

Keywords—T9; trie; regular expression; word prediction

I. Introduction

The T9 (Text on 9 keys) is a word prediction input technology—usually found on mobile phones with 3×4 numeric keypads and in other accessibility technologies—that was originally developed by Tegic Communications (which has since been acquired by Nuance Communications) and was adopted in most mobile phones by almost all major phone manufacturers at the time in the 1990s. The T9 allows users to type in alphanumeric characters via multipress which is a text entry system where a set of letters is attributed to a number on a keypad, in which taps of the button would cycle through the letters on a given button.

Though now obsolete, replaced with the newer and more modern capacitive on-touchscreen keyboards with QWERTY layouts, there are still a handful of individuals who are still a devout user of the T9 and multipress for its simplistic design, tactile feedback, and ease of use. One other aspect of this keypad that cannot be overlooked in terms of its software prowess and brilliant engineering—which is proven by how its concept is still used, adapted, and improved upon even in current mobile phones through touchscreen keyboard softwares—is the predictive text algorithm that it has which is a dictionary-based prediction that allows for fast-access through the trie data structure. Not only does the predictive text technology order the predictions based on similarity and alphabetic order, but it also does this through familiarity of certain words, and also features an ever-expanding dictionary, where certain words that have been used over and over again will eventually be added to the built-in trie data structure (user-database, or **UDB**) on a mobile phone.

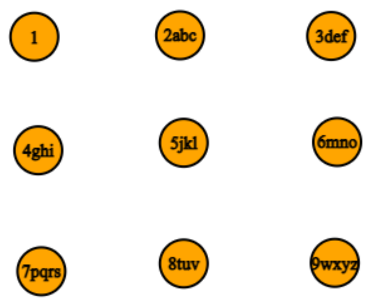


Fig. 1.1 T9 keyboard illustration

Other features of this technology also encompass things such as automatic punctuation as well as clitics for appropriate languages, as well as textonyms which are a combination or order of words that are commonly used, hence increasing the practicality and 'intelligence' of the predictive text algorithm. Though novel and certainly interesting, these additional features will not be further discussed in the contents of the paper.

This paper will solely discuss upon the topic of the barebones prediction algorithm, but through converse methods using regular expressions to the conventional method (trie data structure usage). The paper will also expand upon the alternative method's shortcomings as well as practicality, further developments, and other possible use-cases in various areas of science and technology in the future.

II. Fundamental Theorem

This section will mainly cover the definitions as well as theoretical knowledge needed as a pre-requisite to fully grasp what will be discussed in the following sections. Among other topics, an exposition of the trie data structure as well how it relates to the T9 predictive text algorithm is going to be touched upon, as well as some of the basics and fundamentals of how regular expressions work, along with some other minor subjects that are still related to the main topic.

A. Text on 9 keys

As previously mentioned, the Text on 9 keys, or abbreviated T9, is a predictive text technology found on mobile phones that were used in virtually all mobile phones in the late 1990s.

The T9 stored predetermined words in a data structure that allowed for fast lookup access/retrieval, which enhanced its predictive capabilities, it allowed for word predictions to be done from a combination of single keypresses where the input value would then be matched with the words stored in the data structure. The Text on 9 keys also had a considerably efficient compression rate, at about 1 byte for every word, using an algorithm that was optimized for that specific use case.

To further demonstrate how a T9 would predict certain keypresses, the following word predictions will be shown for specific keypresses.

22737: acres, bards, barer, bares, baser, bases, caper, capes, cards, cares, cases

46637: goner, goods, goofs, homer, homes, honer, hones, hoods, hoofs, inner

2273: acre, bard, bare, base, cape, card, care, case

729: paw, pay, Paz, raw, ray, saw, sax, say

76737: pores, poser, poses, roper, ropes, roses, sorer, sores

B. Trie

A trie is a data structure that is based off the conventional tree data structure of n-ary. The trie or more commonly known as the prefix tree is usually reserved for uses that had the need to store nodes with alphabetical values and most generally used in auto-complete algorithms, data compression, string matching as well as spell checking.

The trie stores sequences of known words inside a tree where leafs of a parent node would mean that a word of those nodes would connect and form one string of word. This is better illustrated with the following image.

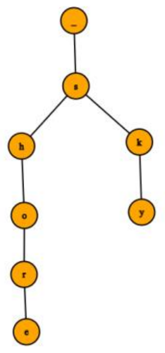


Fig. 2.1 Trie data structure visualised

The trie illustration shown above depicts a trie that has a root node, in this case it is labeled as node '_', and stems to the 's' character which then stems again to the letters 'h', and 'k', which then when assembled as complete strings can construct the words 'shore' or 'sky' or anything in between 'shore' or anything in between 'sky'.

The data structure allows for better time complexity to search for certain words because of how the tree stems into different characters, which bounds certain branches off, if they prove to be mismatched at an early stage, hence eliminating the need of traversing said branch and its children/leafs.

C. String Matching

A string can be defined as a sequence of alphabetic, numeric, symbollic, and glyphic characters. These strings can be further subdivided into its constituent strings, which are called substrings. Suffixes and prefixes exist in the scope of strings, prefixes referring to the substring that ranges from the beginning of a string to a defined cut off point, while suffixes refer to the substring that ranges from a cut off point before the last character, to the last character of the string.

String matching is the process of having an algorithm of choice detect matches or mismatches in a string's pattern when compared to another string.

D. Regular Expression

Regular expressions, commonly abbreviated as regex, is a sequence of characters with syntactual rules that define a search pattern for a string to be matched with. For the understanding of this paper, the readers should be well acquainted with these following basic regular expression rules.

\d	Match a digital number or [0-9]
\w	Match any letter of a-z or A-Z, 0-9, _
\s	Match whitespace
[abc]	Match any one of a, b, c
(abc)	Match abc exactly once
(abc)+	Match abc one or more times
(abc)?	Match abc zero or one time
(abc)*	Match abc zero or more times
[^abc]	Match not a, b, c
\^[abc]	Match starting with a, b, c

III. Methodology

As the heading suggests, the methodology section will cover the specific way an operation in the final imlementation is going to work, by explaining step-by-step the process and logic behind it. The following paragraphs will contain an example of a specific case that will hopefully show the methods

A. *Conceptual Implementation*

The mechanisms in which the T9-like word prediction program will work is relatively straightforward, in the sense that it only requires basic programming knowledge and a preparatory brief of regular expressions to understand.

Essentially, the program will take in input of a numeric string value between 2-9 (mirroring how actual T9 keypads are designed) character by character, allowing for predicted words to change in real time for every time the word is updated. The compounded word comprised of individual characters will then be iterated over and then be translated into a regular expression through a predetermined key-value pair data structure of some sort (note that this data structure will be very miniscule in size, due to how little attribution needs to be done, a sum of eight which is the number of numeric string characters between 2-9), which will then create a regular expression version of the original numeric string value input.

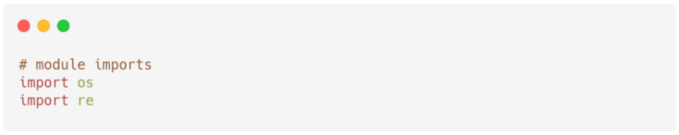
The program will then iterate through all the predetermined words listed in the word dictionary one by one and determining with regular expression evaluation whether there is a match, in which if there is, that word is going to be appended to the matched word list, which will then be printed out for the user to view.

IV. Results and Discussion

This passage of the paper will mainly discuss about the conceptual implementation of the regular expression algorithm (which will cover a brief overview of how the final program/implementation is going to be produced as well as the step-by-step procedure written in descriptive text), as well as the algorithmic implementation of the actual final program that will be discussed in more detail after a short snippet of the code has been shown.

A. *Algorithmic Implementation*

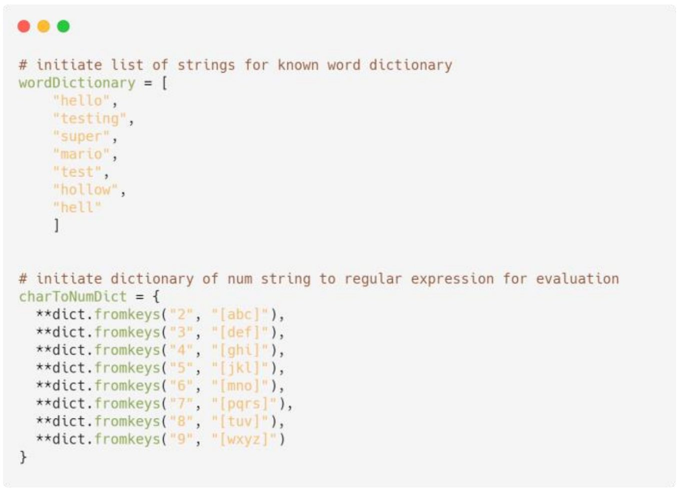
The algorithmic implementation of the method will be done in the programming language Python (version 3.8.9). As a means of explaining the source code of the program in a thorough manner, the program will be divided into its smaller abstractions so that it will be easier to get a grasp of the overall flow of the program.



```python
# module imports
import os
import re
```

Fig. 4.1 Required module imports

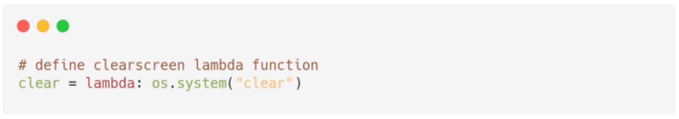
The first pre-step of the program is to import the necessary modules required to construct the algorithm later on, which will differ depending on what programming language the algorithm is built in. The python standard OS module is imported for its clearscreen functionalities which will be implemented in a lambda function later on, while the standard RE module is imported for its regular expression operations capabilities which will be used main bulk of the word prediction function.



```python
# initiate list of strings for known word dictionary
wordDictionary = [
    "hello",
    "testing",
    "super",
    "mario",
    "test",
    "hollow",
    "hell"
    ]


# initiate dictionary of num string to regular expression for evaluation
charToNumDict = {
  **dict.fromkeys("2", "[abc]"),
  **dict.fromkeys("3", "[def]"),
  **dict.fromkeys("4", "[ghi]"),
  **dict.fromkeys("5", "[jkl]"),
  **dict.fromkeys("6", "[mno]"),
  **dict.fromkeys("7", "[pqrs]"),
  **dict.fromkeys("8", "[tuv]"),
  **dict.fromkeys("9", "[wxyz]")
}
```

Fig. 4.2 Initialization of predetermined words and letter attribution

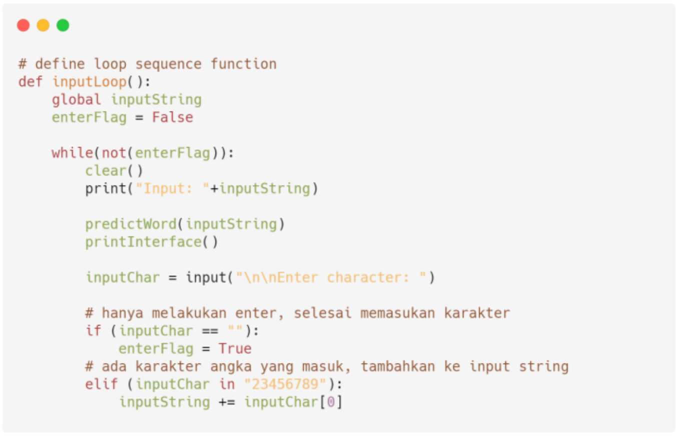
The initialization procedure of predetermined knowledge consists of two parts, one being the word dictionary itself, which is a list of string, easily expanded upon (if the programmer decides to add an expanding dictionary feature) and can also easily import a word list from an external comma-separated or whitespace-separated value file as its built-in word dictionary. The second part is the initialization of the Python dict that contains numeric string value to letter attribution, for regular expression translation later on in the process, this is done by assigning key-value pairs, the keys being the numeric string value while the values are regular expressions of the possible letters attributed to that key.



```python
# define clearscreen lambda function
clear = lambda: os.system("clear")
```

Fig. 4.3 Additional clearscreen function

The clear function above is a lambda function, which does not differ from a regular function, only in the fact that it only has one expression in its body. The clear function calls a method from the OS module to clear the screen, this function will be used later to refresh the command line interface when a new character is input.



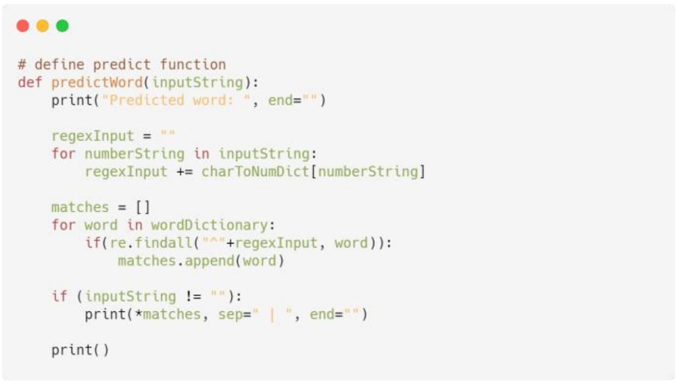
```python
# define loop sequence function
def inputLoop():
    global inputString
    enterFlag = False

    while(not(enterFlag)):
        clear()
        print("Input: "+inputString)

        predictWord(inputString)
        printInterface()

        inputChar = input("\n\nEnter character: ")

        # hanya melakukan enter, selesai memasukan karakter
        if (inputChar == ""):
            enterFlag = True
        # ada karakter angka yang masuk, tambahkan ke input string
        elif (inputChar in "23456789"):
            inputString += inputChar[0]
```

Fig. 4.4 Input loop sequence

The input loop function covers the majority of the program except for the supplementary exit features written in the main code snippet which will be shown in the following paragraphs. The input loop consists of referencing the global inputString variable, which is initially an empty string, then setting the initial boolean value of the enterFlag (the flag which will be set to true once a user enters without putting in a character, signaling that an inputString reset ought to be done) to be false.

The while loop runs while the program still has not met an empty string-enter yet and will then continue the body which comprises of a clearscreen (so that the screen refreshes every time a new character is input) and print current inputString sequence and will then move on to do the word prediction using the predictWord function (will be explained in the following passage below) and print the multi-press keypad interface as visual aid.

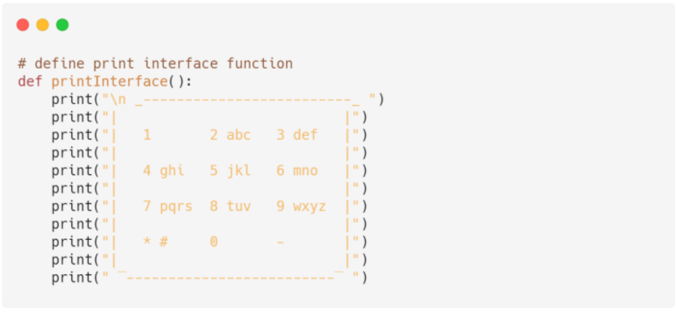
It will then move on to do a check whether or not the input character is an empty string or not, on the case that it is, the enter flag is going to be set to true and the flow of the program is going to exit out of the loop, and on the other case that it is not, it will stay in the loop, adding the latest character input to the inputString which will then be reprocessed.



```python
# define predict function
def predictWord(inputString):
    print("Predicted word: ", end="")

    regexInput = ""
    for numberString in inputString:
        regexInput += charToNumDict[numberString]

    matches = []
    for word in wordDictionary:
        if(re.findall("^"+regexInput, word)):
            matches.append(word)

    if (inputString != ""):
        print(*matches, sep=" | ", end="")

    print()
```

Fig. 4.5 Word prediction algorithm

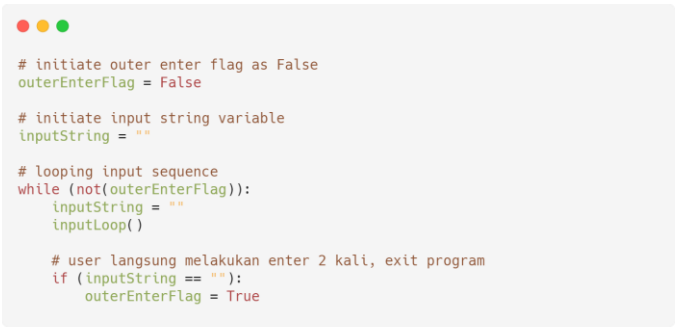
The predictWord function is the main highlight of the entirety of the program, since it contains the part of the regular expression string matching algorithm which has been discussed before. The program will iterate through the characters of the numeric string value, then will then translate it into a regular expression through the dict which has previously been defined. Then the program will go onto finding word matches in the word dictionary/list by iterating through the whole list and finding if the word matches with the aforementioned regular expression. It will then print the results of the matches separated with the vertical bar glyph ('|').



```python
# define print interface function
def printInterface():
    print("\n _-------------------------_ ")
    print("|                            |")
    print("|   1       2 abc   3 def    |")
    print("|                            |")
    print("|   4 ghi   5 jkl   6 mno    |")
    print("|                            |")
    print("|   7 pqrs  8 tuv   9 wxyz   |")
    print("|                            |")
    print("|   * #     0       -        |")
    print("|                            |")
    print(" ¯-------------------------¯ ")
```

Fig. 4.6 Interface print function

The printInterface function is one of a relatively simpler kind as it merely acts as visual aid and does not add any functional benefits nor features to the program.



```python
# initiate outer enter flag as False
outerEnterFlag = False

# initiate input string variable
inputString = ""

# looping input sequence
while (not(outerEnterFlag)):
    inputString = ""
    inputLoop()

    # user langsung melakukan enter 2 kali, exit program
    if (inputString == ""):
        outerEnterFlag = True
```

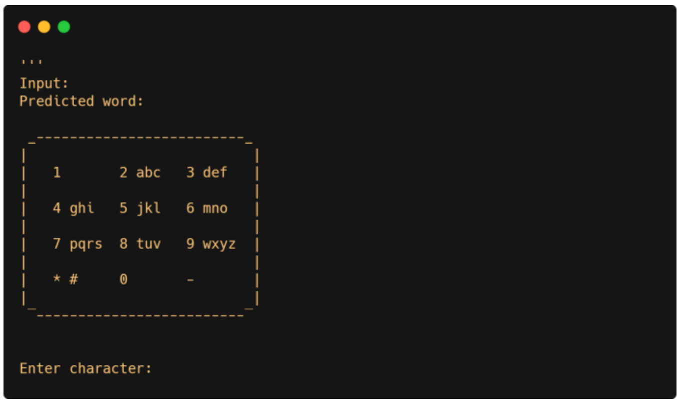
Fig. 4.7 Main program

The main program does a few things in procedural order, first being, setting the initial value of the outerEnterFlag (that will be used to determine whether the user would like to terminate the program by immediately inputing an empty string to a newly reset inputString) to false, and initializing the inputString for use.

The while loop runs as long as the outerEnterFlag is still false, which will only be set to true if the user would like to terminate the program or exit out completely. The while-loop body will consist of the reinitialization of the inputString to be an empty string (to help in reseting the inputString after the program has exited out of an inner input loop), then calling the inputLoop function.

B. *Algorithm Output*

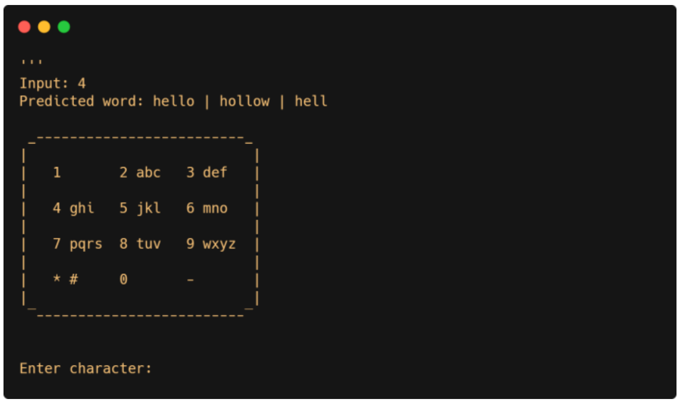
To simulate a case of the T9 keypad being typed in real time, snippets of the code output at certain points will be shown along with its discussion and explanation. The following program tested can be constructed by simply assembling all the aforeshown snippets of code sequentially and running the test case showcased in the following passage.



```
'''
Input:
Predicted word:

 _-----------------------_
|                         |
|  1      2 abc   3 def   |
|                         |
|  4 ghi  5 jkl   6 mno   |
|                         |
|  7 pqrs 8 tuv   9 wxyz  |
|                         |
|  * #    0       -       |
|_                       _|
  ------------------------

Enter character:
```

Fig. 4.8 Starting output

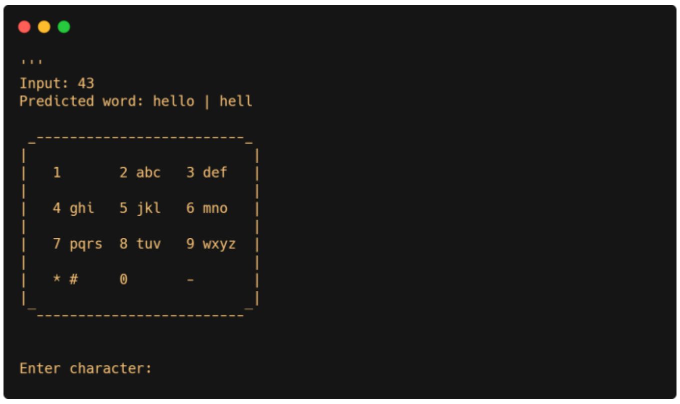
The above output is shown right after the program is run, every part of the interface print sequence is done, and only the character input is left to do by the user. A user will then be able to input any numeric string character of choice between 2-9 (with 2 being attributed to a, b, & c, and continues on for the rest of the numeric string values).



```
'''
Input: 4
Predicted word: hello | hollow | hell

 _-----------------------_
|                         |
|  1      2 abc   3 def   |
|                         |
|  4 ghi  5 jkl   6 mno   |
|                         |
|  7 pqrs 8 tuv   9 wxyz  |
|                         |
|  * #    0       -       |
|_                       _|
  ------------------------

Enter character:
```

Fig. 4.9 Output after an input of '4'

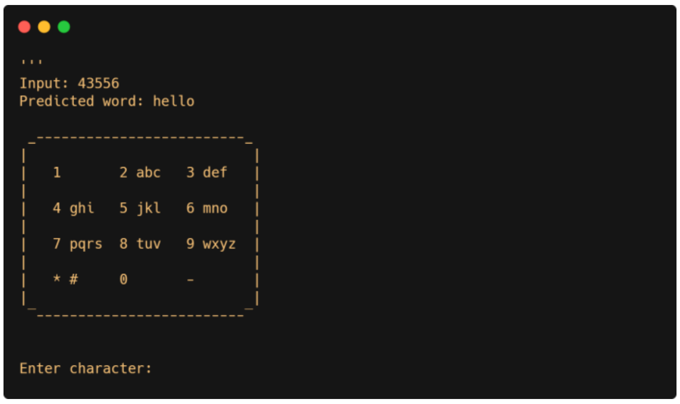
It can be seen that when the user inputs the character 4, the program outputs the predicted word hello, hollow and hell, this matches with the word dictionary defined in the snippets explained previously. The numeric string value of four is attributed to three different letters, namely g, h, and i. The program will convert the numeric string value into the regular expression [ghi] and will then search for any first letter matches in the dictionary, in which hello, hollow, and hell match (due to all of the words beginning with the letter 'h'), and as the dictionary does not contain any other words beginning with the letter g or i, this shows that the list of predicted word shown is sufficient and hence deemed correct.



```
'''
Input: 43
Predicted word: hello | hell

 _-----------------------_
|                         |
|  1      2 abc   3 def   |
|                         |
|  4 ghi  5 jkl   6 mno   |
|                         |
|  7 pqrs 8 tuv   9 wxyz  |
|                         |
|  * #    0       -       |
|_                       _|
  ------------------------

Enter character:
```

Fig. 4.10 Output after input of '43'

Using the same line of logic in the previous paragraph, it can be seen that the word 'hollow' have been eliminated/shaved off from the predicted words list, this is due to the fact that the user has inputed the numeric string value of '3' which means that the only possible combination of words when written in regular expression is [ghi][def], and this proves to be still true for the word 'hello' and 'hell' which both start with the characters 'he'.



```
'''
Input: 43556
Predicted word: hello

 _________________________
|                         |
|   1      2 abc   3 def  |
|                         |
|   4 ghi  5 jkl   6 mno  |
|                         |
|   7 pqrs 8 tuv   9 wxyz |
|                         |
|   * #    0       -      |
|_                       _|
  -----------------------


Enter character:
```

Fig. 4.11 Output after input of '43556'

The last output snippet of the program showcases another similar case, where one of the previously predicted words is shaved off, due to the input string having one more character than the previously predicted word character length. The word 'hello' is still predicted though, due to the fact that the numeric string value of '6' is attributed to the regular expression of [mno] which matches the letter 'o'.

V. Conclusion

To conclude the findings of this paper, the use of regular expressions to do the string matching of a barebones version of a T9-like word prediction algorithm is indeed very feasible and viable to do especially at this point in time, given enough computing-power resource, and storage, and/or execution time.

The alternative method of string comparison shown in the contents of the paper (comparing all the words contained in the predetermined dictionary/list of words against a numeric string input value which is then converted to a small list of predetermined regular expressions—which comes from a numeric value's possible equivalence with the alphabetic characters it is attributed to on a multi-press phone keypad/keyboard), which directly opposes how the conventional T9 string comparison method works (comparing the numeric string input value against the pre-constructed trie data structure which contains all the predetermined words of the dictionary), can be said as a novel approach, though the definition of 'novel' that it stands by is one that is best described by the word 'unusual' instead of 'groundbreaking'. This is due to the sheer inefficiency of the alternative method which essentially runs through a whole dictionary of words to retrieve word predictions, instead of traversing a pre-constructed tree, where branches of words that already have been mismatched will no longer be visited further (heavier emphasis on preprocessing, but by far more efficient and user-friendly).

That being said, much more can be done towards the implementation to better the quality of the final resulting program. Optimisations can be done towards the alternative string-matching method, namely using data structures like Python's dicts or Javascript's literal object notation, to simulate a branching tree, although this optimisation would prove to defeat the whole purpose of using regular expressions if the ending data structure resembled too much of a tree as well. From a feature addition perspective, other supporting hallmarks of the original predictive text technology can be added such as the user-database expanding dictionary, which can be easily added to the current algorithm by simply adding every inserted word to the list of predetermined words. Among other features, the addition of a word familiarity-based sort would also be fairly simple and straightforward as a naïve approach would only require for the word dictionary to store a tuple of a word and a default-familiarity level of 0, which will then increase by one every time a word is typed.

The author hopes that the contents of this paper incited sparks of inspiration and the joy of exploration of various computer science topics in the respected readers and would like to end the paper off in an open-ended note as to inspire exploration of more interesting uses of predictive text algorithms that were shown.

Video Link at YouTube

The following URL is a link to a youtube video explaining the topic, methodologies, results as well as discussions of this paper, in a much less formal and scientific manner, as to promote public attention and awareness on the novel subject of this paper.

https://youtu.be/Ov_VOWs7Pms

Acknowledgment

First and foremost, the author would like to express gratitude towards God for the opportunity to write this particular paper. The author would also like to thank all lecturers, professors, and aides of the "IF2211 Strategi Algoritma" course at Institut Teknologi Bandung that have been involved in the pedagogical process of instilling the discipline of various important computer science algorithms to the author. The author would also like to specifically thank Dr. Nur Ulfa Maulidevi, S.T., M.Sc. who is assigned to be the author's lecturer in the Algorithm Strategies course of code IF2211, as well Dr. Ir. Rinaldi Munir, M.T. whose learning resources have been heavily used and contributed a very large amount towards the process of formulating and writing this paper. Among others, the author would also like to thank a handful of individuals from his group of colleagues who have helped him in proofreading parts of his paper. The author realizes that there are still many unrefined and imperfect parts of the paper and would like to apologize for any uncalled mistakes present in the literature.

References

[1] Munir, R. (2005). Strategi algoritmik. *Sekolah Teknik Informatika dan Elektro, Institut Teknologi Bandung, Tech. Report*.

[2] Munir, R. (2007). Diktat Kuliah IF2251 Strategi Algoritmik. *Institut Teknologi Bandung*.

[3] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, *11*(6), 419-422.

[4] Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., & Di Pietro, A. (2008). An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review*, *38*(5), 29-40.

[5] Dunlop, M. D., & Taylor, F. (2009, April). Tactile feedback for predictive text entry. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 2257-2260).

[6] Willard, D. E. (1984). New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, *28*(3), 379-394.

[7] Wiegand, K., & Patel, R. (2012, June). Non-syntactic word prediction for AAC. In *Proceedings of the Third Workshop on Speech and Language Processing for Assistive Technologies* (pp. 28-36).

[8] James, C. L., & Reischel, K. M. (2001, March). Text input for mobile devices: comparing model prediction to actual performance. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 365-371).

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 09 Mei 2021

[illegible]

James Chandra

James Chandra - 13519078