Comparison Between Brute Force and Trie in String Matching for Autocomplete

Muhammad Fadhriga Bestari (NIM 13516154)
Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha No. 10 Bandung 40132, Indonesia
fadhriga.bestari@gmail.com

***Abstrak*— Autocomplete is arguably one of the most familiar features today that is being implemented in a lot of applications such as Google and Line, two of the most used applications in Indonesia by almost everyone. Autocomplete is so commonly used that now it has become one of the things that people expected from any given application that offers string input from it's users. The most important thing to be aware of in implementing autocomplete is how fast the system needs to determine what the user is currently trying to type. Autocomplete that is not being solved in a reasonable timeframe should not be used since it will not benefit the user in any way. In this paper, author will explain how trie data structure will improve the performance of a system in autocomplete problem.**

Keywords—string matching; trie; brute force; autocomplete

I. INTRODUCTION

Technological advancement produces various new easier ways for people to operate their tasks. Automation of daily tasks is one of the reasons why people continue to innovate and develop technology. Tasks that needed heavy liftings such as moving boxes now can be done with a push of a button. Loading and unloading cargoes without the help of machines is near impossible, and now, with the help of automation people don't need to operate the machine personally anymore.

Although automation is used in a lot of different fields, automation in a more general sense that most people use is string autocomplete. Autocomplete is a very subtle feature that we can find almost everywhere, from Google to our favourite texting applications. It is so widely used that sometimes goes unnoticed and most people don't realize how badly they depend on it.

Implementing autocomplete feature, though, is not an easy task. In this day and age, most people are good enough typers that slow autocomplete won't cut it anymore. Imagine having to wait three second before the program manages to return possible strings that you are currently typing. If an autocomplete can't produce any result in a reasonable timeframe, it can't be used. This is why implementing an algorithm that produces the fastest result with the smallest complexity possible is needed.

II. THEORY

String matching is a process of finding certain patterns in a given string or text. There are many methods specifically developed to achieve the most efficient way to match a string with a pattern. Knuth-Morris-Pratt and Boyer-Moore are two examples of algorithm that are developed to tackle string matching problems. However, there also exists algorithm such as Brute Force algorithm in string matching problems that is easier to implements than other algorithms.

A. Brute Force Algorithm

String matching that utilizes Brute Force algorithm is mostly straight forward. As the name implies, Brute Force algorithm exhaust every possibility to find the best result from a given problem. It is usually the first and easiest method to solve any given computing problem, although most of the time it will not return the result in a reasonable timeframe.

In string matching, Brute Force operates as such :

1. Treat a given pattern and string as an array of caharacters
2. Compare every character in string with every character in pattern
3. If the i-th character in string is the same as the j-th character in pattern, then traverse pattern, else traverse string.

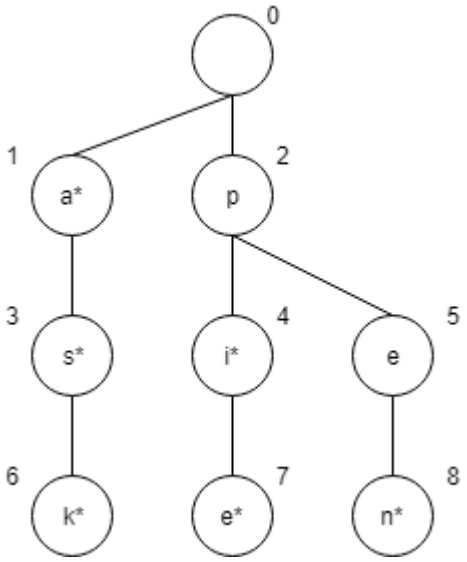
Brute Force algorithm has a complexity of O(n.m), where n is the length of the string, while m is the length of the pattern. It is relatively slow, especially if the length of the string is substantially greater than the pattern, or both of the string and pattern have a lot of characters in them.

B. Trie Data Structure

Trie is a different more sophisticated way to string match. Trie, pronounced "try", comes from the word re**trie**val. It is spelled similiarly to how tree is spelled, and it actually operates as such. Trie, in a nutshell is also considered a tree data structure.

Trie is used primarily for trees that stores characters. The nodes in trie represents the letters of alphabets. Each nodes of letters points to other nodes of letters. Every word that exist in

trie data structure is the collection of a given node's is its parent nodes combined with its own letter. This, in turn facilitates a retrieval of words by traversing down a branch path of the tree.



Picture 1 : Example of trie data structure

In the example shown in picture 1, we have a trie with an empty root which has references to it's children nodes. To implement a trie, each node consists of :

1. Value
2. Array of pointers to children nodes, which means characters that starts after the collection of all its parents nodes
3. Boolean to determine if a given node combined with the collection of its parents nodes is a complete word, indicated by (*)

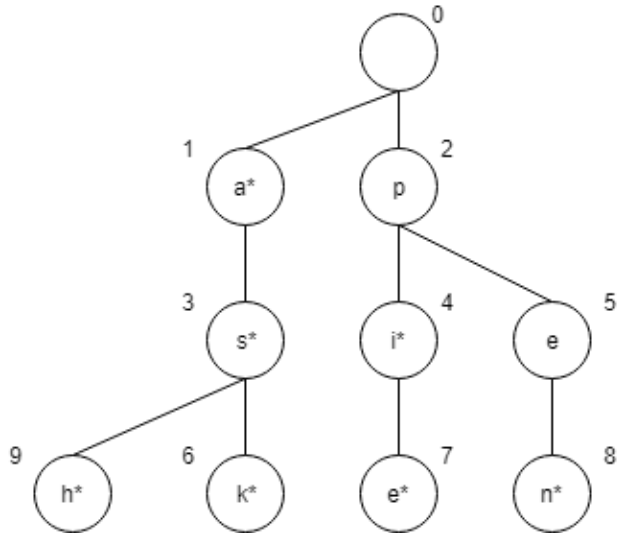
In accordance to picture 1, we can see that there are multiple words that represented in the trie : **a**, **as**, **ask**, **pi**, **pie**, **pen**. Observe how a boolean to determine whether a node is a complete word or not, because all three child node of the root that starts with **a** is a complete word. Boolean can be ignored if all the words in trie cannot continue from the trie's leaf as all leaves would indicate an end of a word.

Notice how a in trie data structure there can be multiple nodes that originates from the same parent. **Pi, pie**, and **pen** all have **p** as their parent. As a result there not need be the same amount of nodes as characters in a given list of strings. The more words that share the same prefix, the fewer nodes needed to be generated.

Trie data structure also change the process of adding new strings to a database. To add a new word to the list of words represented in the trie in picture 1, do :

1. Traverse down the branch where the word should exist
2. If the node doesn't exist, create a new node

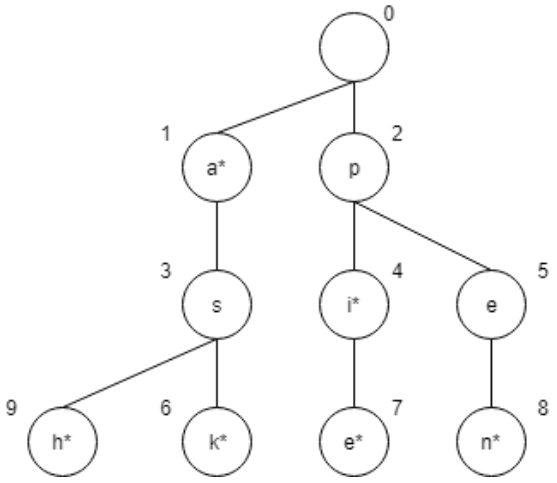
For example, let **ash** be a new string that is inserted into the trie in accordance to picture 1. Start from the root of the trie and determine whether the root has **a** as its child node. Because **a** already exists, traverse down the branch. Next, check whether **s** is a child node of **a**. Because it is, we also traverse down the **s** node. Now, because **h** is not a child node of **s**, we need to create a new node that originates from **s**. Finally, add a boolean in node **h** to indicate that **ash** is a complete word in the trie. The result of **ash** addition to the trie is as such :



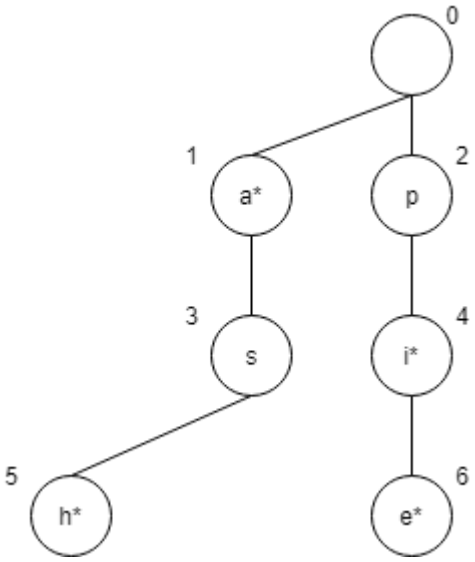
Piture 2 : Addition of new words in trie

Deletion process in a trie data structure also behaves differently than normal. To delete a word from a list of words in a trie data structure, do :

1. Find the node that contains the last character of the word
2. If the node has pointers to other node, simply set the boolean to false, indicating that the node doesn't represent a complete word, else delete the node
3. Move to the parent node, if it doesn't point to any other node, delete that node, else stop.



Picture 3 : Deleting **as,** a word that is also a prefix in other words in trie



Picture 4 : Deleting **ask and pen** from trie

Trie data structude has a complexity of O(n.m) to create, where n is the number of words that exist and m is the longest word in the trie. This is the worst case scenario where all the words doesn't share the same parrent, basically creating the same amount of nodes as characters in the list of words present. To search, insert, or delete from a trie, it needs O(a.n) time where a is the length of word, and n the total number of words.

Notice how it has similar complexity to a Brute Force algorithm, but keep in mind that trie data structure operates in a linear manner with n as the total of strings in the list, whereas Brute Force O(n.m) calculated from matching a single text. So if there were more than one string, to match a given pattern with Brute Force algorithm it will need O(a.n.m) where a is the total number of words, n is the length of the longest string, and m is the length of the pattern.

III. AUTOCOMPLETE PROBLEM

Autocomplete is one of many real world application to string matching program we encounter almost everyday. Autocomplete, although can be used to autocomplete a sentence, such as autocompletes that's implemented in Google and Youtube, here author is going to simplify the problem and mainly focus on autocomplete that operates in a single word only, such as autocompletes found in texting application like Line.

Limiting the problem to a single word drastically decrease the amount of time needed to compute the problem, both with Brute force and Trie data structure. This is due to both algorithm's heavy dependence on the length of the longest string present in the database.

Most applications that implements autocomplete in some form or another will also input the user's commonly used words into the database. The problem discussed in this paper will not account this feature because the patterns used is only the beginning of a word, not the actual word itself. Also, we only focused on autocomplete with exact prefix match, meaning autocorrect will not be accounted for when finding possible words from the database.

Keep in mind that autocomplete emphasizes on how fast a system can determine words that has the same prefix as a given pattern, not where a pattern is located in the string. This is why author use Brute Force algorithm and compare it with Trie data structure, as determining a prefix from a string will be faster with Brute Force algorithm rather than KMP or BM, since both of them would need to create additional data before matching, prolonging its execution time.

A. Brute Force

In Brute Force algorithm, the list of possible words that is available in the database is represented with an array of string. In order to autocomplete a problem, the system needs only to check the prefix of each string. Because this problem is strictly an autocomplete will discount any words that differs ever so slightly from a given pattern.

Due to how different Brute Force stores its available list of words than Trie, author will also measure the time each algorithm create their database respectively. Asuming that each word is inserted one by one, the system would need O(n) time, where n is the total number words that needs to be inserted.

The worst case scenario for Brute Force algorithm that match the prefix of a collection of string will take O(a.n) time, where a is the length of the pattern and n is the total number of words in the database. In Brute Force algorithm, the program has to traverse all possibility before determining the result. The best case scenario for Brute Force algorithm is still take O(n) time, where n is the total number of words.

B. Trie data structure

In order to utilize trie as a method of string matching, first we need to create the trie data structure itself from a collection of words that we will consider as possible words that the user inserted.

To create a trie from scratch, it will take O(n.m) time, where n is the total number of words and m is the longest word in the list of words. Although it is very unlikely that we would reach this number, since realistically, there are tons of words that begins with the same letter.

Nodes in trie data structure will be implemented as follow :

TrieNode :

1. Array of children node with the length of alphabet size
2. Boolean to indicates if the word is a complete word

In order to create a trie, there need be an insert algorithm that's implemented as follow :

Insert :

1. Traverse each level of trie sistematically
2. If supposed node is not yet created, create new child

node

3. If supposed node is a prefix of an already existed word, mark node as complete word

Now to determine whether a word exist in a trie, we need to implement a search algorithm as follow :

Search :

1. Convert the first character of the pattern into ascii
2. Traverse the root the next node that it referenced and repeat for the next character in pattern until the end of pattern is reached
3. If the pointer to a character does not exist, then the prefix does exist in the database

Notice how author does not implement value in the trie node. This is because we only concerned ourselves with whether or not a the pattern exist in the list of text in the database, and all value in the nodes in this particular trie is null.

A node's value can be used in autocomplete problems as a mean to indicate how likely it is for someone to type in that particular word. When implementing a real autocomplete program, with a trie that spread out into a lot of branches, there will be hundreds of possible words that say, starts with the prefix **as**, such as **ask**, **aspalt**, **ash, asia**, **asus**, **ascii**, **ascott**, etc. Determining which word is most likely is currently being typed can be very useful to the user because it will be inconvenient for the user if an autocomplete show all possible words.

This type of implementation also shows how a single node actually has an array of pointers to other node that has the length of alphabet size. This algorithm works similiarly to how last occurance table works in Boyer-Moore algorithm in string matching. Most of these pointers will have a null value, since from the total of 26 of available letters in the alphabet, there will only be 5 to 10 character that any given word can expand towards.

After creating the trie data structure, all is left to do is to search a given pattern inside the trie. Searching a prefix from a list of words with trie will take O(a.n) time, where a is the length of the pattern and n is the total number of words.

C. Brute Force and Trie data structure

If we do a direct comparisons between the two algorithm just from what we have previously discussed, it's difficult to clearly see which algorithm is better. Both of these algorithm needs O(a.n) time to compute, due to how both algorithm needs to, in the worst case scenario possible, traverse all the words in the database.

One thing to keep in mind when determining an algorithm is how different algorithm works better in certain environment than other algorithm. Big O notation is used to calculate the worst time possible for an algorithm to compute a given problem. It does not, however, reflect the average nor the expected compute time for any given problem.

In the best case scenario, it will take Brute Force n amount of checking to autocomplete a given pattern, because Brute Force algorithm needs to check all words at least once no matter what, whereas trie will only compute once to determine if such prefix exist in the database or not, if the first letter of the prefix searched is not referenced by the root. Trie data structure also works similarly to how perm algorithm works, where due to how it traverse down the trie, most of the time trie would not need to traverse all possible node.

Imagine a trie with the same structure as that of picture 4, let the prefix be **a**, then the trie would only traverse the words that starts with **a,** therefore, it would only return **ask**. Trie would not need to traverse the node that starts with **P**, reducing the search time, in this case, by half in regards to Brute Force algorithm. Time reduction in search time may vary, but it is not outrageous to asume that on average, it trie search time would be twice as faster as Brute Force's time.

IV. Experiment results

In this section, author will show the difference between using Brute Force algorithm and Trie data structure in tackling autocomplete problem. This experiment is primarily done to highlight the difference in execution time between the two algorithm. The program is developed and tested in Windows 10 and Core i7 – 6700 HQ CPU @2.60 GHz with 16 GB of RAM.

To determine the faster algorithm, we need to see how both algorithm perform with the same testcase and the same database. All tests are done with 2000 randomly generated words in the database. There are two tests conducted in total. The first execution time result is the sum of data structure creation and prefix matching in said data structure. Whereas the second test is just the execution time needed for each algorithm to search a given prefix from list of word.

Table 1 : Trie data structure accounting data structure creation time

Prefix	**Execution Time**
a	0,011035 s
pu	0,011029 s
sup	0,009021 s
trie	0,00855 s
freak	0,008984 s

Table 2 : Brute Force algorithm accounting data structure creation time

Prefix	**Execution Time**
a	0,006014 s
pu	0,006982 s
sup	0,006014 s

trie	0,006016 s
freak	0,006022 s

Table 3 : Trie data structure search time

Prefix	Execution Time
a	0,001002 s
pu	0,00014 s
sup	0,00001 s
trie	0,000001 s
freak	0,000001 s

Table 4 : Brute Force algorithm search time

Prefix	Execution Time
a	0,00205 s
pu	0,00204 s
sup	0,00301 s
trie	0,002018 s
freak	0,002005 s

Table 5 : Brute Force and trie comparison

	Trie	Brute Force
Average Data Structure Creation Time	0,0095234 s	0,00288234 s
Average Search Time	0,0002012 s	0,0022246 s

V. Result analysis

In the world of programmers, the term "Brute Force" is usually associated with slow algorithm that should be pushed aside for other more comprehensive algorithm. The result of the first experiment, however, where author calculate the total run time of the program for both Brute Force and Trie data structure surprisingly shows that Brute Force actually outperform a Trie data structure in term of overall speed.

In accordance to table 1 and 2, the difference between searching prefix that rarely words starts with, such as "pu" and "sup", the increase in performase almost doubled between Trie data structure and Brute Force algorithm. Also, prefix that a lot of english words originates from, such as "a", shows that Brute Force algorithm still manages to came out on top. Table 1 and 2 shows how a Brute Force algorithm manages to perform better than other algorithm in some situations.

For the second experiment, we calculate only the time it takes for each algorithm to search and determine whether a string matches the given prefix. As expected, Trie data structure search time is significantly lower than that of Brute Force's. The consistency of the experiment result shows that the use of Trie data structure generally will always give faster result.

The experiments also shows how taxing a trie data structure can be. To create a trie that consist of 2000 different randomly generated words, we will need 0,0095234 second, in contrast of 0,00288234 second needed for a regular program to fill 2000 different array index with a string. That is 230% increase in time consumed just to create the data structure needed to compute the prefix matching. However, consider also that a trie data structure is more taxing memory-wise, it decreases the search time needed to determine a prefix in a list of word by 91%.

The significant increase of computing time that needed for trie in creating its data structure is due to how every node has to create an array of pointers with the same exact length as the total letter available in the alphabet. Author specifically used english alphabet for ease of use and author's familiarity with the alphabet. The experiments also did not take account the use of numbers and symbols. Trie data structure needs a lot of memory, significantly greater than that of other data structure, so much so that the machine that author used to conduct these experiments did not allow for more than 2250 words. Inserting each words into an index in an array, however, allow for more words to be inserted into the data structure.

Most autocomplete program are used frequently with a data structure that only needed to be created once and improved as used. It is not unreasonable to say that inefficiency in data structure creation will not cause any noticable problem. Moreover, data structure creation does not necessarily impact how fast a program can determine a prefix in a given list of words in the data structure itself.

In section 1, author explained how an autocomplete program needs to return a result in a reasonable timeframe. In accordance to the experiments results that is shown in the previous section, both Brute Force algorithm and Trie data structure compute in under one hundreth of a second. Consider now if we asume that the data for both Brute Force and Trie is already created, the most time it took to compute 2000 testcase is one thousandth of a second.

Although all four of the experiment conducted resulted in a reasonable timeframe for an autocomplete program, we need to keep in mind that the total number of words that author provided only consist of 2000 words. The english vocabulary consists of hundreds of thousands words, which all needed to be inserted in the data structure for a proper autocomplete program to operates.

VI. Conclusion

Data from the experiments that previously shown indicates that the use of either Brute Force algorithm or Trie data structure, is very dependent on the situation. Dismissing brute force and opting to use other algorithm without proper examination is obviously an ill advised action.

Brute Force algorithm benefits from its use of arrays to store a single word in each index, whereas Trie data structure benefits from its use of pointers to other nodes making it easy to retrieve any data just by traversing its branch from node to node.

Although in accordance to the data experiments in section 4 it is possible to use Brute Force algorithm to be used in autocomplete program, it is generally better to use Trie data structure, due to how much of a decrease in total search time if the database only created once and the autocomplete program is used frequently enough.

Trie data structure as a whole is a great data structure intended for ease of data retrieval. As mentioned before, trie is a tree like structure that works similarly to hash table. This is due to how each word stored in trie can be viewed as a key, and because every node in trie can store a value if implemented, it will work similarly to how keys work in a hash table. The versitality of trie data structure is what makes it a very good way to store data, where it is especially true when the data stored is a collection of words, waiting to be retrieved from the database to be processed.

Acknowledgment

Firstly, author would like to thank Allah SWT for His blessigs, so that author has the strength and delligence to finish this paper wholeheartedly. Without His blessings, author would no have been able to finish this paper properly.

Author also would like to express author's deepest gratitude to Dr. Masayu Leylia Khodra, S.T., M.T., Dr. Nur Ulfa Maulidevi, S.T., M.Sc., and Dr. Ir. Rinaldi Munir, M.T. for the help and care they have given author for the past year in class IF211. Without their guidance, author would not have the knowledge to write this paper.

Lastly, author would like to thank author's friends and family for their undying support, and how they have helped me through thick and thin for the past year and especially this past month during the making of this paper.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifforn Stein, "Introduction to Algorithms, 3rd Edition (MIT Press)".

[2] Dan Gusfield, "Algorithms on Strings, Trees and Sequences : Computer Science and Computational Biology, 1st Edition".

[3] Munir, Rinaldi "Slide of IF2211 : Strategi Algoritma, Pencocokan String (String Matching)".

[4] https://www.cs.cmu.edu/~avrim/451f11/recitations/rec0921.pdf, accessed on May 10, 2018.

[5] https://algs4.cs.princeton.edu/lectures/52Tries.pdf, accessed on May 10, 2018.

Declaration

I hereby certify that this paper is my own writing, neither a copy nor from another paper, and not an act of plagiarism.

Bandung, May 13, 2018

Muhammad Fadhriga Bestari - 13516154