Aho-Corasick algorithm

(A detailed Explanation)

Info:

This research paper is our semester project for Design and Analysis of Algorithms

By:
Moiz Zubair
Sheroz Khan
Afnan Shakeel

Contents

Introduction:

Our group has always been fond of trying to look to work on projects that are going to actually help us gain valuable learning and skills that are required when we land our first job in the industry. We have carefully chosen all our projects to help us achieve understanding of the topics which are essential and may or may not have been covered in our course curriculum. Some examples include Blockchain technology, All shortest path graph algorithms, Radio Frequency Identification (RFID) technology, etc. When we first learnt about our DAA Project we started the hunt to look out for something similar that sparked our interest. In the initial searching phase, we came across many common problems such as DNA Matching, Knapsack algorithm, Graphs (Already done in previous semester), etc., solutions of which were already available however we were really looking for something new, something unique. Luckily, I, Sheroz, happen to follow this YouTube channel which conducts mock coding interviews for jobs at Google carried out by an ex-Google employee. (Mihailescu, 2020) While watching this video, I first learnt about the world of string-matching algorithms and the Aho-Corasick algorithm in particular. To me this seemed fairly new and useful hence I immediately shared the link with my team members and we ended up deciding to work on this project for the reasons already mentioned in the Project Proposal however I'll highlight a few here again for a reference point: The Aho-Corasick algorithm happens to have gazillions of use cases and hence is one of the most used algorithm, its time complexity analysis is really fascinating how it achieves that smooth linear time complexity even after all the complex stages involved, it also helped us dive a bit into the theory of automata which will eventually help us in having a head start when we take that course and lastly it uses trie, one of the most common data structure yet often missing from the course curriculum.

The main purpose of carrying out this project was to understand the different types of string-matching algorithms, how they differ from each other and why the Aho-Corasick algorithm is one of the most commonly used algorithm. We also wanted to be able to thoroughly understand and deploy the Aho-Corasick algorithm wherever needed. We wanted to also dive deep into understanding how state machines work and how the trie data structure plays an essential role in dictionaries and storing strings. Lastly, we wanted to understand how expert level complexity analysis is carried out by exploring research papers written on the complexity of the Aho-Corasick algorithm. If we ended up with some more extra time, we wanted to also look for improvements that could be made to this already amazing algorithm.

Timeline of the project:

1. Exploring the different types of string-matching algorithms on surface level (1 day)
2. Exploring and understanding the Aho-Corasick algorithm. In this phase, we used the initial research paper that was published in 1975. (Alfred V. Aho, 1975)(2 days)
3. Complexity Analysis of the Aho-Corasick (2 days)

4. Reading research papers for improvements, special cases and further strengthening our disconnected concepts to finally understand all the intricate details (1 days)

In stage 1, we used Wikipedia as our primary source. (algorithm, 2021) In stage 2, we used the original paper published in 1975 along with this extremely helpful YouTube videos and this research paper. (Moshiri, 2020) (Alfred V. Aho, 1975) In stage 3, we again referred back to the initial paper published and also some other helpful online resources. (Alfred V. Aho, 1975) In stage 4, we looked for some research papers and went through them to further aid our understanding. (Saima Hasib, 2013) (Sun Liangxu, 2012) We spent approx. 2-3 hours each day.

Contributions

String-matching algorithms:

We started off our research phase by carefully reading this wiki page to understand the different types of string-matching algorithms which include (algorithm, 2021):

Naïve string search:

It is the most basic form of string search in which we check each place, one by one, to look for another string. This is a very inefficient way to perform string search.

Finite-state-automaton-based search:

In this approach, we avoid backtracking by constructing a deterministic finite automaton (DFA) that recognizes stored search string. These are expensive to construct but once we have done the preprocessing it makes our task significantly faster. Initially we weren't really able to understand what a DFA is but later when we read research papers, we did get an idea of what it is and how it works. I will show and explain it later in the document.

Stubs:

Algorithms such as Knuth–Morris–Pratt (KMP) computes a DFA that recognizes inputs with the string to search for as a suffix.

Index Methods:

Faster search algorithms preprocess the text. After building a substring index, for example a suffix tree or suffix array, the occurrences of a pattern can be found quickly. (algorithm, 2021)

Besides these types, string-matching algorithms can also be classified by the way they look for a pattern in a string such as prefix first or suffix first search. They can also be differentiated on the number of patterns they can identify like a single pattern or an array of finite number of patterns.

Aho-Corasick algorithm:

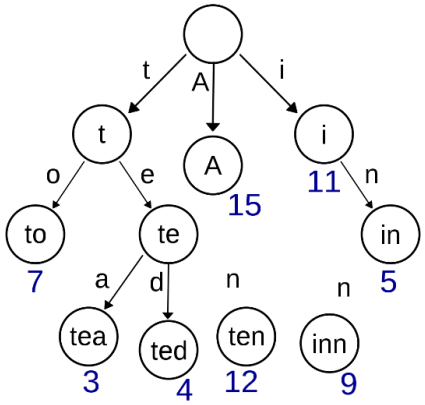
Once we had a thorough understanding of how these algorithms worked, we wanted to classify our Aho-Corasick as one of these. We came to know through research that the Aho-Corasick is a prefix first search and it is also based on constructing a DFA with failure links to avoid traversing one node more than once and backtracking. We also came to know that the Aho-Corasick algorithm is an extension of the KMP algorithm which also runs in linear time like the Aho-Corasick but the difference between that two being that while the KMP can only look for the occurrence of a single pattern in the given string, the Aho-Corasick takes as input an array of finite set of patterns and simultaneously finds the occurrences of all of these in linear time which is astonishing to say the least.

The Aho-Corasick algorithm uses the trie (Pronounced as try) data structure at its core to store the input array of words. For this reason, we started off with understanding what trie actually is and how it works. This video was a great staring point and gave us a good insight as to how data is stored inside a trie structure as nodes containing a map pointing to another node + some characters and the end of word: Boolean which states if a certain node is marking the end of a word. We also understood certain trie operations such as insertions, search and deletion through this very video. Now we will explain what a trie is:

Trie:

A trie is essentially a prefix search tree which is used to store words consisting of characters (Stored as nodes). It consists of various nodes each containing some characters and pointing to another node and we read a word starting from the root and traversing the node towards the bottom till we reach a node that has the end of word Boolean as true stating that this marks the end of a word that is stored in this trie. Following is a trie diagram taken from this wiki page which consists of words array = {to, tea, ted, A, ten, in, inn}. (Trie, 2021)

Note: Don't be confused as for some weird reason two edges pointing to **ten** and **inn** are missing. Also, the numbers are stating the states which you shouldn't be bothered by right now as they will be explained later.



Let's rewind a bit and go to an intermediate stage while insertion is being carried out in the trie. The first word inserted is **to** consisting of two nodes **t** and **to**. Now when we insert **tea,** we find that the node **t** already exists so instead of creating a new node we just follow that **t** from the root and then create another node with **te** and then finally create a node **tea** and mark its end of word Boolean as true. In a similar fashion all the insertions are carried out.

Using trie we can efficiently search the trie with **O(n)** complexity where n is the maximum string length i.e., 3 in our case however trie has its own issues as it uses a lot of space it is highly memory inefficient. For each node we have too many node pointers (equal to number of characters of the alphabet). This problem will be addressed later.

Deterministic finite state automaton (DFA):

Once we had the basic knowledge of trie structures it was now time to understand what Deterministic finite state automaton (DFA) are and how they work. For this we referenced to this video and this research paper and got a fair bit of understanding. (Sun Liangxu, 2012) (Moshiri, 2020) Following is a snippet of the Automata taken from that research paper. (Saima Hasib, 2013)

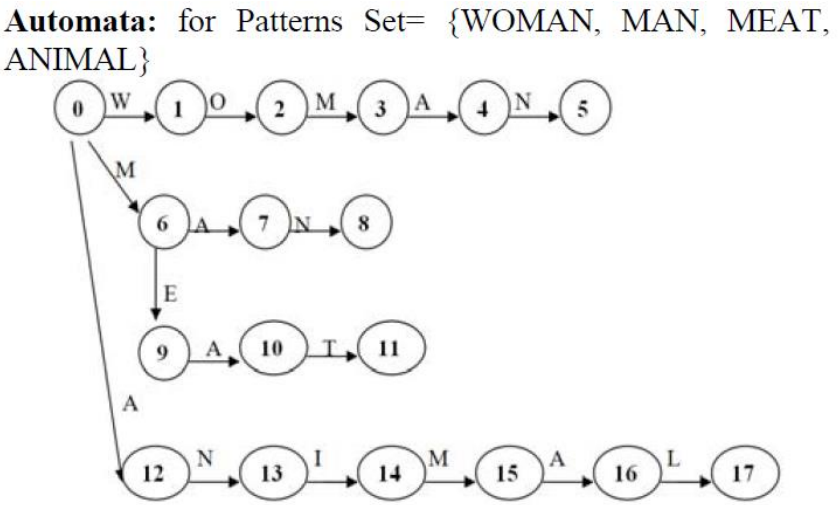


Figure1: Automata

In the snippet what we see is essentially a trie but with states. Root node has **state 0** then for example if we process **W**, we will land on **state 1**. Any word other that **W**, **M** or **A** will always land us back to the root node if we are already on the root node. This is what we call a simple automate. However, our task is not complete yet in order to make sure that while searching we exactly traverse each node only once and do not backtrack that is to go back to root node traversing all the processed nodes again because the nodes are not directly connected to the root node, we introduce the concept of failure links. Failure links are essentially edges that we draw from a node to another node if we find ourselves in such a situation where we want to process a character that is not available in the next node. This is best explained by an example but for the sake of completeness I'll first denote it mathematically as denoted in the original

document. If g is a goto function which allows us to go from state **s** processing character **a**, we denote **g(s, a) = s*** where **s*** is the landing state for example **g(6, A) = 7** while **g(6, E) = 9** from the above figure. Now if we find ourselves in such a situation where **g(s, a) = no such state exists i.e., a fail**, we denote **g(s, a) = fail** for example **g(14, E) = fail** from the above figure, in such a situation we take a failure link which lands us to completely new unexplored intermediate state and we start searching for **E** again from there carrying on from the above example. How these links are calculated will be explained later.

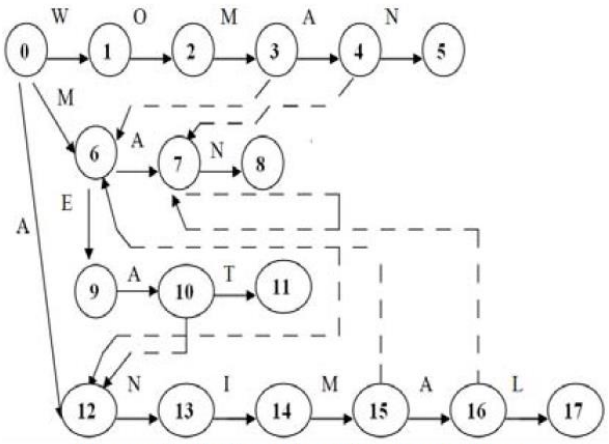


Figure3: Failure function transitions

NODE	FAILURE
0	0
1	0
2	0
3	6
4	7
5	0
6	0
7	12
8	0
9	0
10	12
11	0
12	0
13	0
14	0
15	6
16	7
17	0

Figure4: Failure function table

In the above figure we see all the failure links and each node which does not contain a failure link, its failure link is the root node which is not drawn in the above diagram to keep things simple. Next, we need to talk about the output nodes which are our output functions that output something whenever we reach that node in our machine.

OUTPUT FUNCTION:

Output function gives the set of patterns recognized when entering final state.

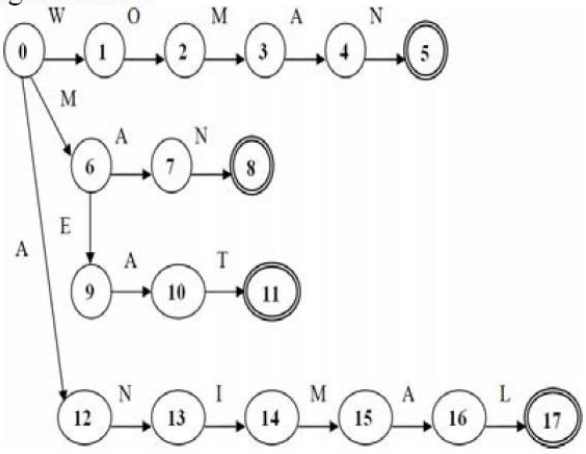


Figure5: Output Function transitions

FINAL STATE	OUTPUT
NODE 5	WOMAN, MAN
NODE 8	MAN
NODE 11	MEAT
NODE 17	ANIMAL

Figure6: Output Function table

So, whenever we reach nodes 5, 8, 11, and 17 we output the recognized words. A trie essentially with states involved, failure links and output function is a DFA, and it is called a machine because it acts as one and everything is defined in it as to what will happen if this or that happens.

Pseudo Code for the Algorithm:

Once we had established what a trie is, what a state machine is and also understood the purpose of the underlying goto, failure and output functions it was the right time for us to take a look at the Aho-Corasick algorithm itself. Below is pseudocode from the original document published in 1975. (Alfred V. Aho, 1975)

Algorithm 1. Pattern matching machine.
Input. A text string $x = a_1 a_2 \cdots a_n$ where each a_i is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function *output*, as described above.
Output. Locations at which keywords occur in x.
Method.

```
begin
   state ← 0
   for i ← 1 until n do
      begin
         while g(state, a_i) = fail do state ← f(state)
         state ← g(state, a_i)
         if output(state) ≠ empty then
            begin
               print i
               print output(state)
            end
      end
end
```

Each pass through the **for**-loop represents one operating cycle of the machine.

The snippet above is what actually the Aho-Corasick algorithm is. It is the simplest implementation that we found online as it allows for abstraction of the functions: goto function **g**, failure function **f** and output function **output** which will be discussed later in the course of this document. We start by setting our state to **state 0** and then process each character of the input **string x** in which we are looking for the patterns using the state machine we built earlier that consisted of characters from our pattern strings array. If a state transition of the goto function results in fail we follow the failure link that is indicated by the **state = f(state)** statement which essentially means to follow the failure link associated with that state. If the transition is

successful, we goto the new state and if it happens to be an output state, we output whatever the node contains else if its empty we just keep processing until all the characters of **x** are processed.

Now once we understood how the Algorithm works, we also had to understand the underlying functions.

Goto Function g:

```
begin
   newstate ← 0
   for i ← 1 until k do enter(y_i)
   for all a such that g(0, a) = fail do g(0, a) ← 0
end

procedure enter(a_1 a_2 ··· a_m):
begin
   state ← 0; j ← 1
   while g(state, a_j) ≠ fail do
      begin
         state ← g(state, a_j)
         j ← j + 1
      end
   for p ← j until m do
      begin
         newstate ← newstate + 1
         g(state, a_p) ← newstate
         state ← newstate
      end
   output(state) ← {a_1 a_2 ··· a_m}
end
```

The above algorithm essentially constructs our Automaton. The **newstate** function specifies every time a new state is made with the state number being assigned to **newstate.** We start of by creating a new state using the statement **newstate = 0** and then loop for each word in our patterns array for example {to, tea, A} and using the procedure enter we enter the characters of each word one by one. Once we are done inserting all the words, we make sure that if any transition tries to happen from the root node where a link is not available for example referring from the above Automaton explained previously, any character beside **M**, **W** and **A** should always result in us staying at the root node. The procedure enter has two parts: the first part keeps following the links from the root node if they already exist for example when inserting tea in the trie the character **t** is already existing in **state 1** as we would have already inserted **to** in the

trie so we just keep following until we reach a node where we find a failure transition. The second part then constructs new node from the node where our transition failed. This way the whole trie/Automaton is established. For reference refer to the trie example given above. Lastly, we also partially construct the output function as in the last state we output all the characters stated by the statement **output(state) = {a1, a2, … am}**.

Failure Function f:

A failure function essentially looks for the longest suffix in the Automaton that also appears as the prefix of some node. For example, if we are at a node **AGH** our failure links can be **GH**, **H** or **root node** considering whatever we can find in the Automaton. It can never be **AGH** as a word only appears once in the trie.

```
Method.
    begin
        queue ← empty
        for each a such that g(0, a) = s ≠ 0 do
            begin
                queue ← queue ∪ {s}
                f(s) ← 0
            end
        while queue ≠ empty do
            begin
                let r be the next state in queue
                queue ← queue − {r}
                for each a such that g(r, a) = s ≠ fail do
                    begin
                        queue ← queue ∪ {s}
                        state ← f(r)
                        while g(state, a) = fail do state ← f(state)
                        f(s) ← g(state, a)
                        output(s) ← output(s) ∪ output(f(s))
                    end
            end
    end
```

The first loop is pretty straight forward as it just add all the depth 1 nodes in the queue and sets their failure links to the root node. The second loop computes the failure links for all the nodes left using the failure links of the previously calculated nodes. It also simultaneously calculates the output functions of the nodes considering the failure links.

Output function output:

The output function is partially computed in goto and failure links, so we do not need another algorithm for it.

Analysis:

Time complexity:

In each run of the algorithm the machine makes zero or more failure transitions followed by exactly one goto transition. Therefore, when processing a string of length **n** there will always be a total of **n** goto transitions and hence the total transitions will always be less than **2n** (goto + failure transitions). The actual time complexity of the algorithm depends on different factors; we could store the goto function in a 2D array allowing for constant access however if the size of the inputs increase significantly it will not be economical rather, we can just store the non-fail values as a linear list or we can just simply store the most used states using indexing allowing for constant access. We can also use binary search trees to store the goto values or simply create a trie (Could be very expensive in terms of space complexity). To store failure values, we just need an array which again allows constant access.

The total number of executions of the statement **state = f(state)** in algorithm 1 is bounded by the total sum of the length of the keywords.

The Algorithm can be divided into three parts: the for loop which runs n (The length of the input string x) times, the while loop contained inside the for loop which is bounded by **m** where **m** is the total sum of the length of the keywords/patterns and the if condition which runs the number of times the patterns appear, we denote it by **z**. Hence to total time complexity becomes **O(n + m + z)** i.e. linear time complexity.

Space complexity:

The space complexity of Algorithm 1 mostly depends on the type of implementation used however if tries are used the space can be very inefficient at times and hence, we will talk about how to improve it later.

Special Case:

We will explain this carefully thought after special case with an example. Let's say our pattern strings contains **K = {a, aa, aaa, aaaa, a^k times}** and our input string is **a^n**. This is where things become really messy. In worst case we'll have to print all the words of **K** at virtually every position of **a^n**. However, even at its worst case it performs better than **naïve search** so it's still better.

Improvements:

As mentioned earlier tries can be really complicated in terms of space hence we started doing some research as to how to tackle this problem. Our research led us to this amazing paper which tries to resolve this issue to a significant extent. (Sun Liangxu, 2012) It was a complex case study and honestly, we weren't able to fully grasp it to its actual potential but essentially what the paper states is to use a different character set to store shift state than the ASCII code. It states "The compressed store modes, banded-row format algorithm and sparse-row format

algorithm, can effectively reduce the memory consumption of automaton." At this stage we don't fully understand how these algorithms work under the hood but if we had more time, we would have loved to explore these too.

All in all our achievement was to fully understand all the intricate details of the algorithm in such a short time frame which makes us believe we did an amazing job as this is **expert level** algorithm as reported by the famous site **geeks for geeks**. However, this achievement meant we had to give our extra time and longer attention span to achieve it.

The problems we encountered was essentially testing ourselves because we have always been video learners living in this age and time but for this project, we had to refer many research papers and articles which was something we wanted to challenge ourselves with because eventually when we go abroad Insha'Allah for our masters we will get into the habit of writing and reading the research papers. We initially also planned to put this document in latex but due to time constraints we decided not to hence time constraints due to upcoming ESEs and other pending projects proved to be the biggest hurdle in exploring more.

Conclusion:

The result of the project is that we are extremely happy for all the effort we have put in learning this difficult algorithm and now if in any point in time we have to explain or use this or any similar algorithms we can easily do so. We also liked how it allowed us to become habitual of reading research papers. If we had to rate ourselves for this project it would be at least a 9/10 leaving that 1 mark for not exploring more improvements but that was due to the time constraints and we would love to work on that in future. We'll try to explore more improvements on the Aho-Corasick algorithms, one that really sparked our interest was the use of GPUs to increase the efficiency and running time of the Aho-Corasick algorithm. This was a research paper we found online but we couldn't read it or document it due to time constraints. We would also love to explore more complex string-matching algorithm and this time also focus on how these are deployed and their practical use cases.

If we could go back in time, we would have challenged ourselves more by exploring the research paper which talk about improvements on the algorithm and their use cases. That is the only objective we would have liked to add in our initial goals however that does not mean our hunt is over we will keep exploring more and keep learning!

References

Alfred V. Aho, M. J. (1975). *Efficient String Matching: An Aid to Bibliographic Search.* Retrieved June 6, 2021, from https://cr.yp.to/bib/1975/aho.pdf

algorithm, S.-s. (2021, May 17). *String-searching algorithm*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/String-searching_algorithm

Mihailescu, C. (2020, September 17). *Clément Mihailescu*. Retrieved from Google Coding Interview With A Facebook Software Engineer: https://www.youtube.com/watch?v=PIeiiceWe_w

Moshiri, N. (2020, April 26). Retrieved from Advanced Data Structures: Aho-Corasick Automaton: https://www.youtube.com/watch?v=O7_w001f58c

Saima Hasib, M. M. (2013). Importance of Aho-Corasick String Matching Algorithm in Real World Applications. *International Journal of Computer Science and Information Technologies, 4(3)*, 467-469. Retrieved from Importance of Aho-Corasick String Matching Algorithm in Real World Applications: https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=4C556F16E36A4581E546097249EF2A37?doi=10.1.1.310.6746&rep=rep1&type=pdf

Sun Liangxu, L. L. (2012). Improve Aho-Corasick Algorithm for Multiple Patterns Matching Memory Efficiency Optimization. *Journal of Convergence Information Technology*, 162-168. Retrieved from https://www.researchgate.net/publication/269780289_Improve_Aho-Corasick_Algorithm_for_Multiple_Patterns_Matching_Memory_Efficiency_Optimization

Trie. (2021, May 19). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Trie