Regular Expression Generation with Brute-Force Algorithm

John Kim
jkim4@oxy.edu
Occidental College

1 Abstract

Regular Expression, also known as Regex, is a sequence of symbols, numbers, and characters that represents a search pattern and/or text manipulations. It is highly versatile and widely supported across many programming languages.

This paper aims to automate the task of generating regex utilizing a brute force algorithm. Given an example of an regex output the code will conduct pre-processing of relevant keywords to simplify the task and find the search-pattern utilizing a brute force algorithm.

2 Technical Background

2.1 Byte Pair Encoding

In natural language processing, tokenization is the process of converting text to smaller interchangeable units called tokens. These tokens can then be re-arranged to form completely new sentences.

Byte Pair Encoding is a well established tokenization algorithm. It finds and merges the most common pairs of adjacent characters in a given corpus until the word reaches a set size, creating tokens that efficiently represent text[5]. BPE is an algorithm designed with Natural Language Processing in mind and did not perform seamlessly as you could expect from a text based corpus. BPE algorithm was utlized in a corpus of regular expressions[4]. Some manual filtering was required afterwards, but it was able to recognize common token pairs such as `.*`

2.2 Standard Deviation

Standard deviation is a statistical measurement that quantifies the amount of variation in a set of values. A low standard deviation indicates that the data points tend to be close to the mean, whereas a high standard deviation indicates that the values are more spread out over a wider range. Standard Deviation was utilized in conjunction with the brute force algorithm. Its purpose is to randomly determine the number of token used in the algorithm with a statistical focus on the mean. The algorithm assumes a mean of 3 tokens and gradually increases the value of standard deviation to increase. Eventually, as the graph nears a flat line, a random number between 1 and 10 is chosen as the token count.

2.3 Regex-based Denial of Service (ReDoS)

ReDoS is a DoS attack caused by algorithmic complexity that drains a given web-services' resources due to it costly time complexity. The time complexity can grow exponentially or polynomially as its input size increases linearly.

Most regex functions have a exponential time worst-case complexity but they are often overlooked during development as they are often, but not always, rarely triggered with genuine inputs. ReDoS failures occur when such regex function is given a maliciously crafted input or a genuine input that triggers exponential/polynomial backtracking. Programming languages predominantly use a backtracking for its regex search algorithm, most notable examples are C#, Java, JavaScript, and Python. Its computational cost is negligible in most use cases, but it can sometimes be detrimental.

There are numerous ways to write a regular expression which gives identical outputs. A regex generated by the brute force algorithm may work but may also be computationally inefficient to the point of overloading the cpu.

2.3.1 2019 Cloudflare Outage

Sometimes a regular expression function can consistently cause a ReDoS failures even on genuine inputs. One such example is the 2019 Cloudflare global outage. Cloudflare is a company that provides a number of internet services one of which is Content Delivery Network(CDN) services. To combat Cross-site Scripting attacks through their CDN, they implemented a regular expression to their fire-wall that detects any Java-Script and HTML code appended to URLs.

```
(?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|
true|false|null|undefined|symbol|math)|
\`|\-|\+)+[)]*;?((?:\s|-|~|!|{}|\|\||\+
)*.*(?:.*=.*)))
```

This was the regular expression in question. The malformed section that caused the outage was:

```
.*(?:.*=.*)
```

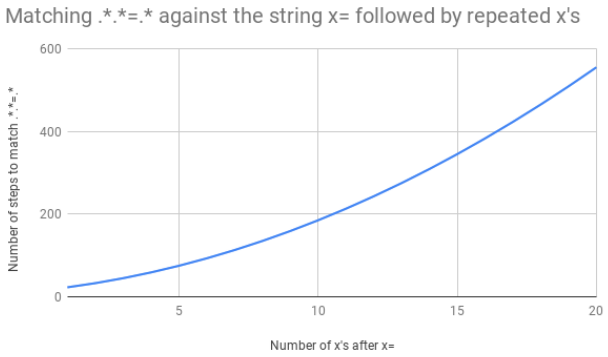

Figure 1: exponential time complexity of Cloudflare's malformed regex

[2]

Now disregarding the non-capturing group:

```
(?:)
```

It can be disregarded since the regex still needs the computational resources to conduct the search and then ignore it. We are left with:

```
.*.*=.*
```

As the simplified regex shows, the first two .* will match greedily and then both would backtrack iterating backwards one unit at a time until it finds an equal-sign or until all possibilities are exhausted. This means that it will take 23 steps to match an input string simple as 'x=x'. 33 steps for 'x=xx' and 45 for 'x=xxx'[2] The complexity is exponential and much worse if the string does not contain an equal-sign as it would need to exhaust all possible combination of the two .* For instance, it will take 4,067 steps to compute a string that is 20 characters long without an equal-sign[2]. Considering how long URLs can get, you can see how this would cause an outage.

The outage demonstrates the disastrous potential of ReDoS failures and how inconspicuous they can be to users. Unfortunately, the program is not able to detect and mitigate ReDoS failures.

2.3.2 Linear Time Regular Expression

Since the 2019 outage, Cloudflare has switched to the Rust regex engine for their regex needs. The Rust engine is deterministic and runs in linear time: $O(N)$.

Deterministic regular expressions such as re2 and Rust regex engine achieves linear time complexity by utilizing a combination of Deterministic Finite Automata(DFA) and Non-Deterministic Finite Automata(NFA), allowing it to avoid backtracking[3]. The primary limitation of such engines is the high memory requirements for complex regexs.

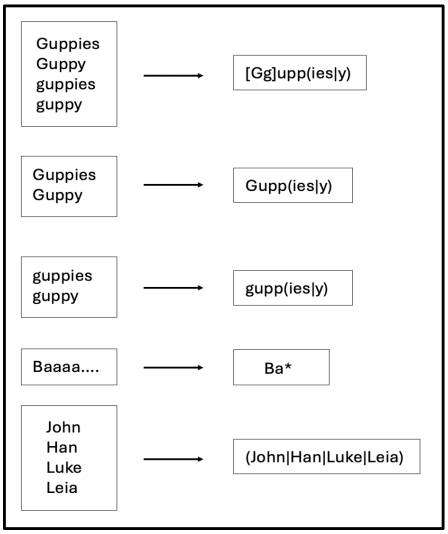

Figure 2: Keyword Pre-Processing Examples

Users are encouraged to use a linear time engine. It is possible for the brute force algorithm generating regex with exponential time worst-case complexity which may lead to a ReDoS failure.

2.4 Prior Work

Prior works in the field of automated Regular Expression generation is limited. This may speak to its difficulty or insignificance even if successfully implemented. I suspect both.

2.4.1 Genetic Programming for Regex Generation

A team of researchers at the University of Trieste in Italy utilized genetic programming to automate regular expression generation as a so called proof of concept. Their findings were limited. Their research did demonstrate an abstract potential for such algorithm. However, it only demonstrated that it's able to utilize genetic programming to generate regular expressions with manual post-processing. Their proof of concept failed to independently generate any coherent regular expression. Furthermore, their algorithm was tested on a very limited sample size of two.[1]

2.5 Methods

The program conducts pre-processing of keywords provided based on the user inputs in order to simplify it as much as possible. Then the brute force algorithm, utilizing both Byte Pair Encoded tokens and Standard Deviation number generation, generates potential regular expressions. Once the correct regex is found, it is then outputted.

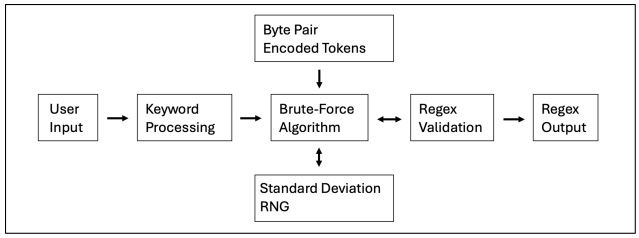

Figure 3: program overview

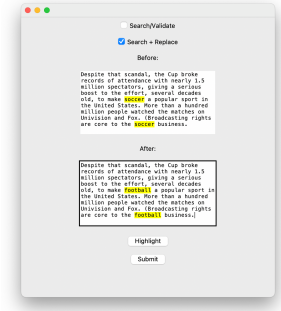

Figure 4: User Interface

The shorter token list is used for the Brute Force Algorithm which contains the most common regular expression tokens and when the list fails to generate the correct regex the complete list is used.

2.6 results

[] The results were averaged from 10 trials. The results show that brute force algorithm is able to consistently generate the correct regular expression when the non-keyword tokens are limited to 5 maximum. The algorithm, however, was not able to reliably generate longer more complex regular expressions.

References

[1] Bartoli, Alberto et al. "Automatic generation of regular expressions from examples with genetic programming". In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 1477–1478. ISBN: 9781450311786. DOI: 10.1145/2330784.2331000. URL: https://doi.org/10.1145/2330784.2331000.

[2] Graham-Cumming, John. *Details of the Cloudflare outage on July 2, 2019*. July 2019. URL: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[3] Groz, Benotıt, Maneth, Sebastian, and Staworko, Slawek. "Deterministic regular expressions in linear time". In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 49–60. ISBN: 9781450312486. DOI: 10.1145/2213556.2213566. URL: https://doi.org/10.1145/2213556.2213566.

[4] innovatorved. *Regex Dataset*. URL: https://huggingface.co/datasets/innovatorved/regex_dataset/viewer/default/train?p=1&views%5B%5D=train.

[5] Zouhar, Vilém et al. "A Formal Perspective on Byte-Pair Encoding". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by Rogers, Anna, Boyd-Graber, Jordan, and Okazaki, Naoaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 598–614. DOI: 10.18653/v1/2023.findings-acl.38. URL: https://aclanthology.org/2023.findings-acl.38/.

	average time(seconds)	average attemtps
`\d[Gg]upp(ies|y)?$`	1.37	126,026.2
`^d[Gg]upp(ies|y)?$`	28.33	2,610,317.7
`\b\D[Gg]upp(ies|y)?\b`	37.57	3,389,479.3
`^(John|Han|Luke|Leia)$`	0.05	4,435.4

Table 1: Average of 10 Trials