DAAR Project 1

Cloning egrep command supporting simplified ERE

Carlo Segat - student number 21115378

Project description

This report describes the implementation of a clone of the egrep command. Given a regex and a .txt target file, the program here described returns the lines in the target file that contain at least one match with the provided regex.
Regexes supported by this program can only contain parentheses, alternations, concatenation, closures, periods (i.e. the universal character) and ASCII alphabetic characters (i.e. {a-z} U {A-Z}).
Javascript is the chosen language for implementation.

Problem definition

The problem we are facing is matching a regular expression against a possibly large text corpus. This is a valuable proposition as all text editors and readers alike need to allow a user to input a query string and show where it appears in the text. Also, computer programs such as compilers have to frequently perform the task of searching text.
A regular expression is a string from a regular language that encodes a textual pattern, a more precise definition can be found here [1]. Since the language of all regular expressions is a regular language, all regular expressions can be encoded as finite state automatons (FSA).
Examples of regular expressions and the patterns they encode are listed in the table below.

Regex	**Patterns**
hello	hello
(h|H)ello	hello, Hello
hel*o	heo, helo, hello, helllo ...
hell.	hello, hella, helle, hellz ...
hello|world	hello, world

Let us stress that our goal is to determine if a string of characters contains somewhere a match with a given regular expression. This complicates matters as it is not sufficient to run the regex once at the beginning of the input.

Analysis and theoretical aspects of algorithms solving this problem

Let us discuss and analyse 3 approaches to this problem:

- FSA construction
- KMP
- Inverse index

The last two approaches are only applicable in specific cases, but given the performance improvement they provide and how common it is for applications and users to search for single words, they are worthwhile.
Note that those approaches are not mutually exclusive and can be combined in practice.
Also note that there are alternative approaches not discussed: for example, we can simulate the regex directly, without pre-constructing the NFA but building it on the fly as we run the regex.

FSA Construction

This approach consists of transforming a "raw" regex to a DFA which can then be used to test if a string matches a regex. Note that for a string of length *n* we may need to run the DFA *n* times before concluding if there is a match or not (i.e. one DFA run starting at each character in the string).

The construction is as follows:

- A raw regex is given, e.g. "(H|h)ello*"
- The syntax tree is produced, the order of precedence of the operators is: closure, concatenation and union; parenthesis can be used to give the highest precedence to the contained expression.
- From the syntax tree, an NFA with lambda transitions is constructed as described in [2]. Essentially we can recursively construct the NFA by following the 4 rules described in the Aho-Ullman book.
- We remove the lambda transitions from the NFA by merging the largest groups of states that can be reached with lambda transitions only. The transitions for one of those new states are obtained by merging the transitions of the old states that make it up.
- We apply subset construction, again, described in [2], to remove the non-determinism and have exactly one destination state per transition.
- Sometimes a so-called "death state" is added. In every state, when we do not have a transition for one of the symbols, we add a transition to the death state to signal that we could not find a match.
- Optionally the resulting DFA can be minimised by deleting equivalent states.

Constructing a DFA from an NFA is $O(2^n)$ where n is the number of states in the NDFA.
One run of the DFA is linear on the size of the input string.
The problem is that, in the worst case, we may need to run the DFA for each character in the string, leading to a runtime of $O(n^2)$.

KMP - Knuth-Morris-Pratt

This approach is applicable only when the regex is only a concatenation of symbols (which is very often the case in practical searches on text). The benefit of KMP is that the runtime is improved from $O(n^2)$ to $O(n + m)$ where m is the length of the regex.
This method does not require building a DFA. It simply checks the current input character (***ii***) with the current regex character (***ir***) and takes an action based on whether they match or not.
For example if we use the regex *"hello"* on the string *"hello world"*, the algorithm would increment both ***ii*** and ***ir*** until matching the last regex character.

The intuition behind KMP is that we can avoid re-running the regex from character *ii + 1* if we failed with a run that started at character *i*. All the information we need to make this decision is in the regex itself and not in the searched string.

As an example consider the regex *"mami"* and the two strings to be searched *"mamà"* and *"mamami"*. For *"mamà"* we would like to not waste time by repeating the regex search starting at 'a', 'm' and 'à' after failing at the first 'a'. On the other hand, we **do** want to repeat the regex search starting on the second 'm' of "mamami" after failing at the first.

KMP gives us exactly this power by computing a carry-over array of length n + 1 where n is the length of the regex (last element always 0 for technical reasons). When we fail matching a character from the regex at indexk **k** with a character from the input string at index **j** we look at the carryover array (CA) at index **k**, if it is greater than 0 we repeat the search with k = CA[k], if CA[K] is less than zero then we need to set j = j + 1 and k = k + 1. When there is a match we simply increment both **j** and **i**.

To fill the carryover array at position *i* we consider all the suffixes of the regex substring that goes from index 0 to *i-1* included. We need to consider the **longest** such suffix that is also a **prefix** of the regex, its length will be the carryover value. The carryover aray construction is often subjected to optimisations and a more detailed construction can be found in the original paper [3].

Inverse Index

This approach is applicable when the query regex is a concatenation of alphabetic characters. The idea is to pre-compute inverse-indexes for common words present in the text. The inverse-index is a word-level inverted index, storing the *line number* at which the string appears.
For example, if the user inputs the regex "hello" we would just need to do one lookup for the word "hello" and find at what line it appears.
To speed up the lookup we could also store the indexed words in an optimised data structure such a radix-tree.
What words are chosen to go into the index depends on the context of the application but a good starting point would be to compute the word list for the document and index the top *n* entries. Alternatively, a list of interesting words can be given (e.g. name of programming languages if the application needs to scan CVs for tech companies).

Details of my implementation

I begin with a preprocessing step. I convert the special symbol '.' into a union of all letters supported ("*(a|b|...|Y|Z)*").
Then I build the syntax tree. I introduced a symbol for concatenation ('.') which is safe because of the universal character dot symbol has been removed during the preprocessing.
I do a pass of the string to enclose all real symbols with parenthesis (i.e. abc becomes (a)(b)(c)) and at this time I count how many real symbols, i.e. letters, there are, if the count is 0 then it means that the regex was malformed (e.g. **| has zero real symbols and it's a malformed regex).

The syntax tree is converted to a NFA in a recursive fashion. I use Javascript classes to encapsulate the logic of a state and the logic of the NFA itself. It would have been more efficient to use a more simple data structure like a 2 dimensional array, but again, if efficiency would have been the main concern I would not have chosen Javascript.

The lambda transitions are removed from the NFA and subsequently, subset construction is performed to ensure the NFA becomes a DFA.

I did not perform the minimisation of the obtained DFA.

The obtained DFA utilisation consists in feeding strings to it (i.e. lines of the book) and the line is returned as soon as one match is found, without looking further.

Tests

Unit tests

I wrote unit tests during the development following a methodology akin to TDD.
For example, the first step is to construct the syntax tree, I do this by considering the operators in order of precedence, therefore I have a test that checks that given a certain string applying the parsing step will result in the expected string, cfg. the image below.

test_syntaxtree.js

```
19  function test_parse_concatenation() {
20    assert(parse_concatenation( string: '(a)(b)', i: 0, left_indices: []) === '((a).(b))')
21    assert(parse_concatenation( string: '(a)(b)(c)', i: 0, left_indices: []) === '(((a).(b)).(c))')
22    assert(parse_concatenation( string: '(a)(b)|(c)', i: 0, left_indices: []) === '((a).(b))|(c)')
23    assert(parse_concatenation( string: '(a)(b)|((c)*)((a)(b))', i: 0, left_indices: []) === '((a).(b))|(((c)*).(((a).(b))))')
24    assert(parse_concatenation( string: '(c)(((a)(b)(c))*)', i: 0, left_indices: []) === '((c).(((((a).(b)).(c)))*))')
25  }
26
27  function test_parse_symbols() {
28    assert(parse_symbols( string: 'a', i: 0) === '(a)')
29    assert(parse_symbols( string: 'abc', i: 0) === '(a)(b)(c)')
30    assert(parse_symbols( string: 'abc|a*|ab', i: 0) === '(a)(b)(c)|(a)*|(a)(b)')
31    assert(parse_symbols( string: '(a|b)|abc', i: 0) === '((a)|(b))|(a)(b)(c)')
32  }
```

End-to-end tests

I tested my program on two corpora of text, “A History of Babylon, from the Foundation of the Monarchy to the Persian” [5] and “The omnipotent self, a study in self-deception and self-cure” [4].

The tests run the egrep command and my program for the same regex in the same file.
Outputs are compared and the tests succeed only if they are identical, i.e. I am expecting the same lines to be returned from the two programs.

Please note that since some characters, such as parenthesis, are not supported by my program, I limited testing to only certain regexes. For example, if we run *egrep “W.*have”* we would get all lines that have a ‘W’ followed by ‘have’ **regardless** of the characters in between. But my program would only match lines that have **supported characters** between ‘W’ and ‘have’. For example, the line *“William, as described in [23] is ...”* will go unmatched because I do not support square brackets.

Performance comparison

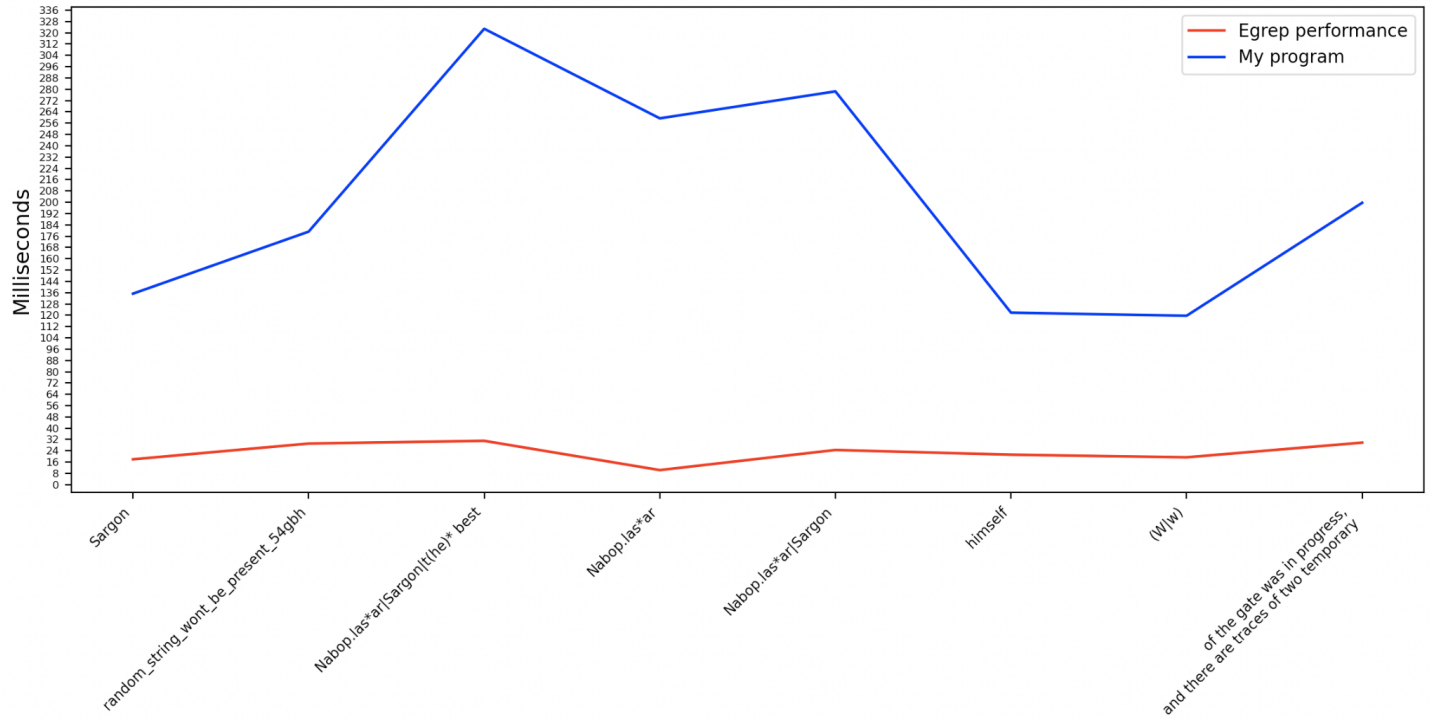
Below we can see pictures comparing the performance of the linux egrep command and my program on the regexes listed on the X axis. On the Y we can see the execution time. For example the regex *“himself”* egrep took around 20 milliseconds to complete the search and return the relevant lines, on the other hand, my program took around 120 milliseconds.

The first picture refers to searches on “A History of Babylon, from the Foundation of the Monarchy to the Persian” [5] and the second to “The omnipotent self, a study in self-deception and self-cure”.

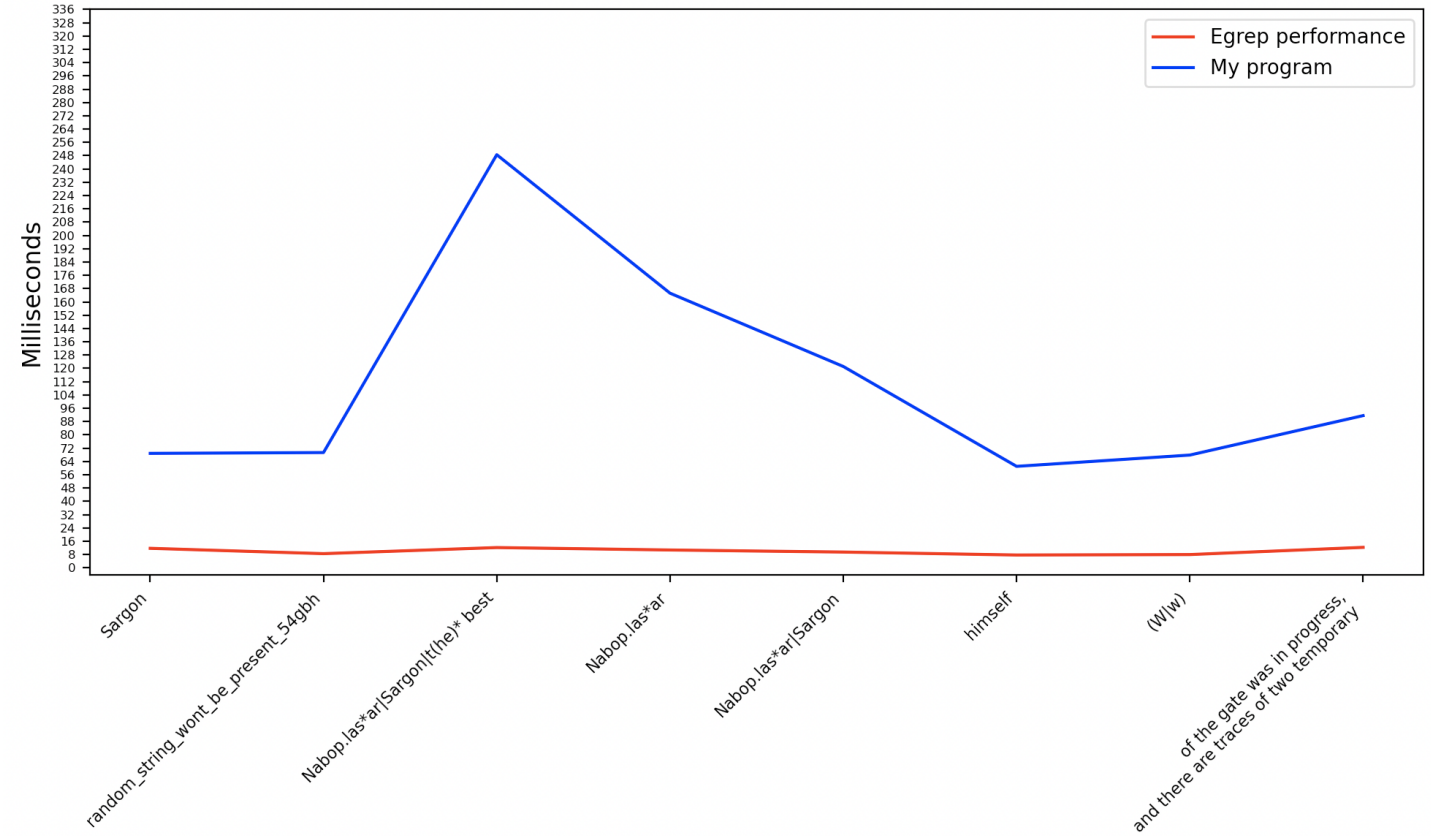
We notice immediately how egrep outperforms my implementation. This is not surprising as egrep is a very sophisticated and mature piece of software that has been finely tuned for decades [6].
Another signal that testifies the quality of egrep is the fact that the performance is not subjected to turbulent changes when using regexes that have different operators or lengths. On the other hand we do observe this variability on my program: simpler and shorter regexes take less time to execute.

We also note a decrease in the absolute values of the execution times when we perform the search on a smaller file: History of Babylon is ~750KB and the other book is ~200KB.
The runtime averaged over all the regexes for egrep is 22.93 ms and 10.04 ms (for the 2 files respectively).
For my implementation, it is 202.01 ms and 111.69 ms.

Performance comparison for History of Babylon.



Performance comparison for The omnipotent self.



Conclusion

Searching a body of text with a regex is a problem in the domain of pattern recognition. There are problems in pattern recognition that require the full power of turing-machines, but for regular expression, given that they belong to regular languages, the power of finite state automatons suffices.
It is fascinating to see how much research went into optimising this task given how common it is to search strings for users and programs alike.
Also, implementing this myself made me realise what must be going on when I use a regex in my favourite programming languages.

Bibliography

[1] The single Unix specification - Regular Expressions - available at https://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html

[2] Al Aho and Jeff Ullman - Foundations of Computer Science, available at http://infolab.stanford.edu/~ullman/focs.html#grad

[3] Fast pattern matching in strings* (1974)
donald e. knuth, james h. morris, jr. and vaughan r. pratt

[4] The omnipotent self, a study in self-deception and self-cure by Paul Bousfield available at https://www.gutenberg.org/cache/epub/66496/pg66496.txt

[5] A History of Babylon, from the Foundation of the Monarchy to the Persian,available at https://www.gutenberg.org/ebooks/56667

[6] The history of grep, the 40 years old Unix command, online article, available at https://medium.com/@rualthanzauva/grep-was-a-private-command-of-mine-for-quite-a-while-before-i-made-it-public-ken-thompson-a40e24a5ef48