Regex Engine Implementation

(Using epsilon-NFA and DFA)

Namrata R
PES1201700921
Gaurav H Sangappa
PES1201700282
4 ‘B’

Introduction:

What are regular expressions?
Regular expressions are a notation for describing sets of character strings.

What is a regular expression engine?
A regular expression "engine" is a piece of software that can process regular expressions, trying to match the pattern to the given string.
It takes in a regular expression pattern in the form of sequences of characters.
The regex engine also takes in a string and checks if the given pattern matches the string.

Applications of regex engine:

1. Grabbing text in files or validating text input when programming in languages such as C, Java or PHP.
2. Many text editors use regex engines for Searching and possibly replacing text in files. Searching and replacing across pages of code when using in an IDE such as Visual Studio, Komodo IDE.
3. Renaming multiple files at a time in an advanced file manager such as Directory Opus.
4. Finding records in a database.
5. Telling Apache how to behave with certain IP addresses, URLs or browsers, in htaccess for instance.

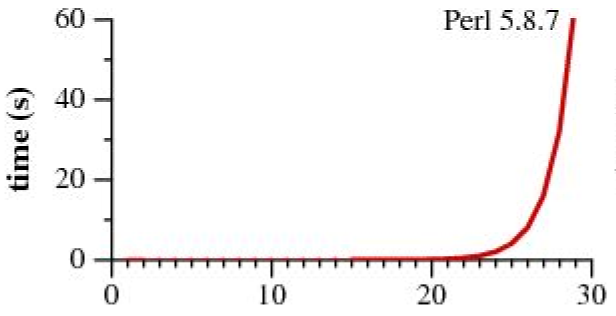
To list a few.

Methods to implement a Regex engine:

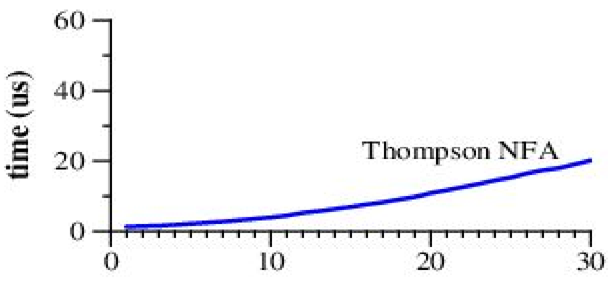
Of some of the many methods, two approaches to implementing regular expression matching are discussed below. One of them is in widespread use in the standard interpreters for many languages, including Perl. The other although more theoretical, might be a better option.

Method 1(Backtracking):
In this method, we backtrack into recursion with the remainder of the string matching to the regex after the recursion fails. Then try all permutations of the recursion as needed to allow the remainder of the regex to match.
In case none of the permutations work, ie, none of the paths are taken, then we can conclude that it isn't a valid string.
This backtracking approach has a simple recursive implementation but can read the input string many times before succeeding. If the string does not match, the machine must try all possible execution paths before giving up.

Method 2 (Implemented in the project):
The more theoretical approach, which we have used in the algorithm is using epsilon NFA and DFA.
This is similar to the known Thompson Algorithm. Here the regex is converted to its respective NFA and the string is checked character by character to see if it can reach the end state of the corresponding DFA on reaching the end of the string



.

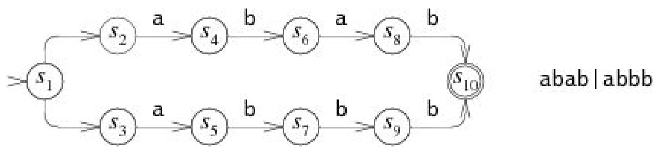


Implementation Of Project

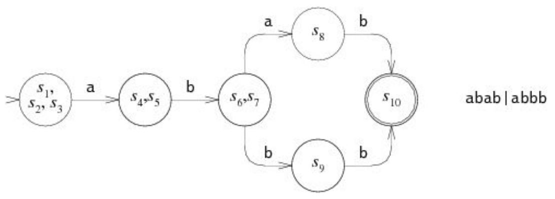
In our project, we have started off by using the same logic as in the Thompson NFA implementation.

1. First the regex is converted to a more systematic format for further computation.
2. Then the respective transitions for the NFA are found. Operators that our regex engine is compatible with are ‘+’, ‘.’,’()’,’?’,’*’,{} and ‘\’.
3. The epsilon closures for each state of the NFA is found.
4. We now store the NFA in the form of an epsilon NFA transition table.
5. This is now converted to a DFA by the standard procedure along with the usage of the epsilon closure.
6. The DFA transitions are stored as a transition table, with new state names, final states (0 state is the common Dead state and 1 is always the start state in this model)

For example, here is the NFA for abab|abbb, with state numbers added:



The equivalent DFA would be:



7. Finally, for the given input string, the generated DFA is traversed character by character. If at the end of the string the DFA has reached a final state then the string is considered to be valid and said to match the regular expression given as input.

Analysis of Method used

Although this method that we have implemented, is lesser seen in the real world and said to be used mainly in theory, as mentioned earlier, usage of NFAs and DFAs for string matching may be a much more efficient method taking into mind the time efficiency.

It is said that Perl could take about a minute to match strings of character size above 30, but for the same string and regex taken, the Thompson NFA method is said to take around a few milliseconds.

Our project is a step towards using this method as a regex engine. Although our project gives more importance to the time factor rather than the efficiency with respect to space.
Using DFAs make it much easier to see if the string matches, rather than using parallel tracking of paths in an NFA to see if the string follows any one of the multiple paths NFA may take.

Our project is a small example to show the working of such an engine, keeping into account most of the operators of a simple regular expression.

Scope for Improvement:

1. In the present implementation, most of the data (example storing transition tables, etc) are done using static variables. We can dynamically allocate them to make it space efficient as well.
2. Currently, some redundant DFA states are being formed in the final DFA. The DFA can be minimized thus reducing the need for additional space.
3. In the NFA being formed, few extra epsilon closures are included. This can further be reduced to increase efficiency.

Conclusion:

Regular expressions are known to play a very important part in language processing and in today's programming world.
Hence it is also important to use an efficient method to compute such regular expressions.
In the assignment project, we have tried to use one of the older approaches as practiced in theory.
It is said that regular expressions are one of computer science's examples of how using good theory leads to good programs.
Our attempt is to come up with an engine similar to this, following its basic features and functionality.
This can always be further extended by adding more testing parameters and generalizing it further.