Theory Of Computation

Project Report

"REGEX to NFA to DFA convertor"

Abstract

This project with objective of "REGEX to NFA to DFA convertor" is made by me for the innovative project submission of Theory Of Computation (CO-303) course for 5th semester of my B.Tech. degree. This project and the whole source code is developed by myself, except for using some open-source libraries such as NodeJS, expressJS, etc.

1 Credits

This project is developed completely by myself as an Individual and is not copied from any other project on knowledge database. I have used the knowledge I have gained in TOC theory class to develop the algorithmic idea of the project and implemented every nook and corner of the project modules by myself. Extra credits to be given to some websites which helped me in getting the basic conversion algorithms I needed and also to open source library NodeJS for helping in ease-of-implementation of my project.

2 Introduction

The objective of my project is "Conversion of regular expression to NFA and conversion of resultant NFA to DFA".

Regular expression is used to denote the strings that can be accepted by a given regular language. Also, the regular language can be denoted by using finite automatons (NFA's and DFA's). The conversion of regular expression to NFA and DFA is possible, and vice-versa. But we only know this conversion manually. So my project is to automate this conversion. I have created a web application for this project, where the user gives a regular expression as input and the application converts that regex into respective NFA and DFA.

3 Technologies and algorithms used

I have used only open source technologies for the purpose of ease-of-implementation and cross-platform compatibility reasons. Thus the project I created is virtually free-of-cost and can be used for both development and deployment purposes by anyone. The link to project I uploaded on Github is given below:

The technologies I have used in this project are:

1. HTML, CSS, JavaScript (for front-end development)
2. NodeJS, ExpressJS (for back-end server creation)
3. Visual Studio Code (for text editor)
4. FireFox (as browser for development)
5. Some VS code extensions (to speed up my coding capabilities)

I have used basic algorithms for conversion from REGEX to NFA and then conversion from NFA to DFA. The theory that helped me in its implementation and the websites from where I studied them are given below:

1. Conversion from REGEX to NFA (https://www.geeksforgeeks.org/regular-expression-to-nfa/)
2. Conversion of E-NFA to DFA (https://www.javatpoint.com/automata-conversion-from-nfa-with-null-to-dfa)

However, I don't recommend using this project as it is for production purposes as it is not tested thoroughly for production reliability and is merely created for innovative project submission purpose.

4 Software requirements and Installation

The project source code is JavaScript written and thus requires a Web browser to run. Any latest web

browser would suffice, but Firefox is preferable as it is tested to run on Firefox itself.

As for libraries and dependencies, the user should have:

- Node Package Manager (NPM)
- A terminal

That's all. The dependencies could be installed using NPM so its straight forward.

The steps of installation and running the web application are:

- Extract the zip file of source code to an empty directory.
- Open a new terminal in that directory.
- Type command " npm init -y " to initialize the project as a Node project.
- Type command "npm i" to install any required dependency, that is not pre-installed.
- Type command "npm start" to run the program. The program will open in your default browser automatically.
- If on your browser, it shows an error, just refresh the tab it should load up the program.

5 Description of Front-end part of app

The front-end of the application I have developed is a one-page application which includes the instruction on how to write regular expression in input, examples of good and bad regular expression (bad ones will give either error or unexpected results). Then there is an input block for regular expression and a submit button which will initiate the conversion.

Once the user enters the regex in input and click on button, the regex is converted to NFA, then the NFA is converted to DFA and both NFA and DFA are shown below the input block.

The screen shot of front-end page is given along in the report, for reference.

6 Description of back-end

I have kept the back-end as simple as possible. The back-end consists of a node-JS file "server.js" which a basic template for hosting a local-host server and serving the static files (All HTML, CSS and JS files of front-end).

The server hosting is managed by Express module which when a default get API call is made, redirects the client to my HTML page. The server is set up to listen on PORT 5050, which can be changed by editing the PORT variable's value in "server.js".

7 E-NFA class(also used for DFA)

The NFA and DFA implemented in my code is created by making a class template for Enfa and using the same with some manual modifications for DFA.

The class Enfa has given members:

1. This.states: Array of states present in the finite automata
2. This.inputs: Array of inputs present in the F.A
3. This.initial_state: data member determining the initial state of F.A.
4. This.final_states: Array of states which determines all of the final states in the automata.
5. This.transition: A nested object containing details of all of the transition present in F.A. The transition can be seen by "this.transition[from][via] = array of states in given transition".

There are many member functions in this class templates, which are used to define basic property of F.A. as well as some utility functions used for achieving objectives of this project.

The member functions are:

1. Sow_transition(): Util function for dev purposes
2. Add_transition(from, via, to): Function to add a new transition having "FROM state", via input, "TO state" parameters
3. add_transition(from, via, to)
4. Get_transition(): Util function for dev purposes
5. Klene_plus(): Applying Klene plus on current E-NFA
6. Klene_closure(): Applying Klene closure on current E-NFA
7. Get_state_val_with_offset(): Util function for dev purposes
8. Concat(): Function to apply concatenation of two NFAs
9. Addition(): Function to apply or operation between two NFAs i.e. result = NFA(1) | NFA(2)
10. Sigma_closure(): Function to get sigma_closure of given NFA
11. Rename_state(): Function to rename a state of NFA
12. Add_dead_state(): Function to add dead state (to be used for DFA)

8 Conversion from Regex to NFA:

I have divided conversion of regex to NFA into three pars for modularity. These parts are:

1. Modify_regex(): Modify regex for irregularities(it doesn't completely check for irregularities as it is out of scope of this project)
2. Infix_to_postfix(regex): Convert the Infix regular expression to postfix regular expression
3. Solve_for_nfa(regex): Solve postfix regex to get resultant NFA

The first part is to convert the input regex into a regex without irregularities. It is not a very deep modification, rather just the addition of concatenation wherever there is no expression between two characters. For example, it will convert (abc) to (a.b.c). I have not implemented deep modification as it a multi-step complex process and out of scope of this project, as my main objective is the conversion of a *correct* regular expression to NFA and DFA.

The second part converts the infix regular expression to a postfix regular expression. This part is an innovation of mine as I couldn't think of any method to convert a normal regular expression directly to nfa. Also, I wanted this project to have a part of innovation. The conversion of infix regex to postfix regex is somewhat similar to infix-to-postfix conversion of an arithmetic expression, with some modifications. The precedence of operations is replaces by precedence of keywords for regex, which can be obtained from the internet. The precedence of brackets is similarly kept the highest. Fortunately, this conversion gives us the correct answer in the form of NFA at later stage.

The third part is conversion of postfix regex to NFA. The steps in this conversion are:

1. If you get a character, create a NFA for the single character and push the NFA to the stack.
2. If you get a keyword (any one from * | . +):
    a. If the keyword is '*':
        i. if the stack is empty, return failed conversion(-1)
        ii. else pop the top from stack, apply Klene closure and push back to stack
    b. If the keyword is '+':
        i. if the stack is empty, return failed conversion(-1)
        ii. else pop the top from stack, apply Klene plus and push back to stack
    c. If keyword is '.':
        i. if the stack has less than 2 elements, return failed conversion(-1)
        ii. else pop two elements apply concatenation and push the result in stack.
    d. If keyword is '|':
        i. if the stack has less than 2 elements, return failed conversion(-1)
        ii. else pop two elements apply NFA(1)|NFA(2) and push the result in stack.
3. Check the stack, if it contains exactly 1 element, return it as result, else failed conversion.

This two-step conversion is a success and I have tested this conversion for about 100 regex manually. There were some bugs I found during testing, which I corrected. The current conversion process is flawless without any bugs.

.

9 Conversion from NFA to DFA

The conversion from NFA to DFA I have implemented, is the basic conversion from E-NFA to DFA using the sigma closure method.

I have taken reference from “Javatpoint” website. I have included the link in *references* as well as *Technologies and algorithms used* unit.

The steps of conversion from NFA to DFA are:

1. Copy the symbols(Terminals) of NFA to DFA
2. Create a DFA using NFA class(we will manually manage the difference between NFA and DFA)
3. Get the sigma closure of NFA in an array "arr"
4. Create two arrays, totArray: to hold array of "Stringified set of states", curArray: to hold array of "set of states" not processed for further branches
5. Add initial "Set of states" to curArray and its Stringified version to totArray
6. Rename the initial state of DFA to Initial "Stringified Set of states" of NFA
7. While curArray is not empty:
    a. Take first element of curArray and name it as "from".
    b. For each of the input symbol (except sigma) of the NFA:
        i. For each of the state in "Set of states" of FROM:
            I. Get all of the reachable state from the given state via given input, and create a "set of states".
            II. If the resultant "set of states" is not included in totArray, add it in totArray and curArray.
            III. If the resultant "set of states" is not empty, add a transition in DFA from "FROM set of states" via given symbols to "TO set of states".
            IV. Remove the "FROM set of states" from curArray, as it is now processed.
8. Rename the complex states of DFA to simpler states
9. Add dead state to DFA, if required

I have implemented renaming of DFA states and adding dead state to the DFA for ease of understanding of the user.

10 References

The references for my project are:

1. https://www.geeksforgeeks.org/regular-expression-to-nfa/
2. https://www.javatpoint.com/automata-conversion-from-nfa-with-null-to-dfa
3. https://www.npmjs.com/
4. https://code.visualstudio.com/
5. https://expressjs.com/
6. https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode