**PROJECT DESCRIPTION**

This project converts regular expressions into their Finite State Machine equivalents. And from the DFA’s, the program would be able to test the regular expression pattern against many different strings. The program would first read in a regular expression and parse it. From the parsed expression, it will generate an NFA, which will then be converted into a DFA. The DFA could then be made into its minimal state form. In its DFA state, the regular expression would be able to test itself against different strings.

**RESEARCH & DESIGN**

The algorithms that this program utilized came from the textbook that was used throughout the course[1]. The algorithm for minimizing a DFA is specified in the book and was even part of an assignment in the class. Along with this, so did the algorithm for removing non-determinism from an NFA. Even the general idea as to how to convert a regular expression into an FSA came from the book.

**DFA Minimization**

var machine = MachineUtilities.ParseFileToDFA(file);

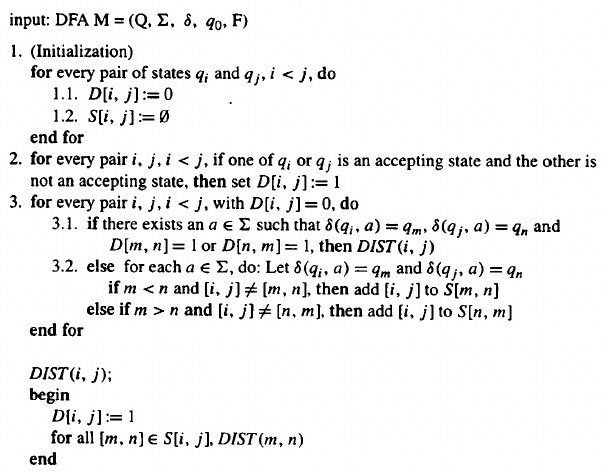
var diff = machine.GetDistinguishableElements();

var minimized = machine.Minimize(diff);

*Figure 1: Sample code from the program that reads in a DFA from a file and then converts it into a minimum state DFA*

The first part of the program implemented was the DFA minimizer. The algorithm starts off by determining which states are equivalent by constructing a matrix D that determines the equivalence amongst different states.

If two states, qi, qj ϵ Q, are distinguishable, then the value D[i, j] is set to true. States qi and qj are distinguishable from one another if one state is an end state while the other is not. Another case in which two states would be distinguishable is if for δ(qi, u) → δ(qk, u) and δ(qj, u) → δ(ql, u), where u ϵ Σ and qk, ql ϵ Q, qk and ql are distinguishable from one another. If the states are found to be distinguishable, then D[i, j] is set to true. The algorithm iterates through each pairing of i and j several times. It stops when an iteration makes no changes to D.



*Figure 2: Algorithm 5.7.3 from the book[1], which determines the equivalent states of the DFA*

States qi and qj are equivalent if after the matrix-building algorithm runs, D[i, j] = false. Also, they are then consolidated into a single state.

As you can see in my sample code from Fig. 1, the matrix D (referred to as diff in my code) is derived from the DFA. The DFA then returns a minimized copy of itself by taking in values from the matrix.

var m = MachineUtilities.ParseFileToNFA(file);

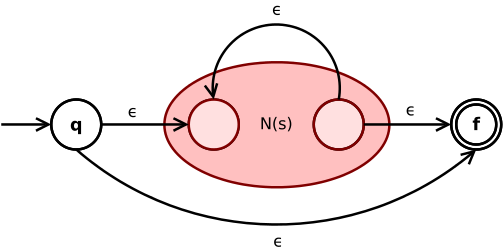
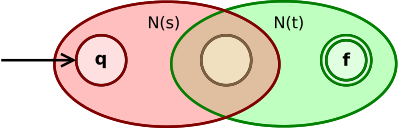
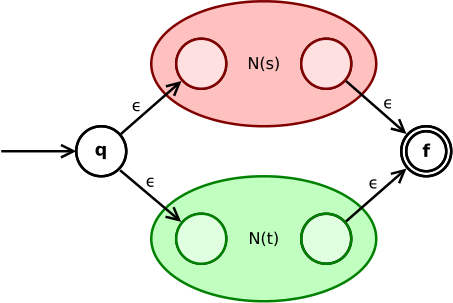
var tTrans = **new** TFunction(m);

var mDFA = tTrans.ConstructDFA();

*Figure 3: Sample code from the program showing how a DFA is generated from an NFA*

**NFA to DFA Converter**

Also defined in the book is an algorithm on how to convert an NFA into a DFA. First, a λ-closure table would be constructed that shows where states can get to when given λ as an input string. From there, a t-table is constructed that adds in the closures for the elements onto the transitions. A new transition, referred to as Ø, is defined which denotes that a transition leads to an empty set. Once the t-table is created, the DFA is then built starting from λ-closure(q0). As the algorithm expands a state, it expands on each of it transitions in the t-table. During this expansion, a new node is created for each unique combination of states in the transition. This goes on until each state has been expanded.



*Figures 4, 5, 6: From source[3], these are the different archetypes that comes with encompassing an NFA with a larger one. From top to bottom: union, concatenation, Kleene star*

**Regular Expression Parser**

The regular expression parser is composed of the parser itself and a visitor. While the parser goes through the regular expression and generates a sort of derivation tree, the visitor travels through the created tree and processes the data.

The grammar for which to parse the regular expressions by is based on a grammar found at this source[2], which is meant to process Perl-style regular expressions. The grammar is defined in such a way as to accept Kleene star (‘\*’), plus (‘+’), concatenation, union (uses ‘|’), and sets. A set in this case can be identified as a string in between two matching brackets (‘[‘ and ‘]’), which defines a union amongst all the characters in that string. For example, “[abc]” is equivalent to “(a|b|c)”. The grammar is also limited to just accepting alphanumeric characters.

When visiting the parsed tree, there are certain fundamental regular expressions that can be transformed into NFA machines. For example, a set or a single character can be turned into a machine. These machines could then be a part of a more encompassing regular expression by applying an NFA archetype. The archetypes are defined for Kleene star, union, and concatenation. Visual representation of this can be seen in Figure 4.

**IMPLEMENTATION**

Pattern ((a\*)(b\*)(c\*)): (a\*)(b\*)(c\*)

NFA:

δ a b c

>q0 a:[q1] λ:[q1]

q1 λ:[q1, q0, q2]

q2 b:[q3] λ:[q3]

q3 λ:[q3, q2, q4]

q4 c:[q5] λ:[q5]

\*q5 λ:[q5, q4]

DFA:

δ a b c

q0 q0 q0 q0

>\*q1 q1 q2 q3

\*q2 q0 q2 q3

\*q3 q0 q0 q3

Minimized DFA:

δ a b c

q0 q0 q0 q0

>\*q1 q1 q2 q3

\*q2 q0 q2 q3

\*q3 q0 q0 q3

Input Tests:

δ(q1, "aabc") –> δ(q1, "abc") –> δ(q1, "bc") –> δ(q2, "c") –> δ(q3, λ) ACCEPT

δ(q1, "bc") –> δ(q2, "c") –> δ(q3, λ) ACCEPT

δ(q1, "ccb") –> δ(q3, "cb") –> δ(q3, "b") –> δ(q0, λ) REJECT

*Figure 7: Converts the regex “a\*b\*c\*” into an NFA, then a DFA, and finally a minimum-state DFA. The DFA is then tested on a series of strings. Note that the DFA cannot be minimized further.*

The project was implemented using the C# programming language and it was compiled with mono. The regular expression parser is auto-generated using ANTLR, which only allows for left-recursive grammars. A jar file containing the ANTLR program and a dll linking to the ANTRL runtime library is included within the project so that no further installations are needed. Mono, however is required to run this program.

**TESTING & SAMPLES**

Within the *demos* directory in the project, there sits a series of demos in which the user can run. Each demo contains a table that defines an NFA or a DFA to be tested with. The program was tested against the questions from the book[1] and exam questions. One such example is question #4 from exam 2, in which you are asked to remove the non-determinism of an NFA that is equivalent to regex “a\*b\*c\*”. A sample runtime of this application is shown in Figure 7.

**RESULTS ANALYSIS**

The goals of this project was to be able to convert a regular expression into its DFA equivalent. This has the intermediate steps of parsing a regular expression into an NFA, removing non-determinism from an NFA, and producing a minimum-state DFA. As shown above, all of these pieces have been implemented.

**USER GUIDE**

When compiling the program, make sure that you have the latest version of Java to run the ANTLR file and have a version of C# installed to compile the program itself. There are make commands to compile and run the program and to clean up the local repository, which can be seen in the make file in the project’s root directory.

Simply typing in “mono program” will run a few hand-picked examples that shows off the capabilities of the program. The same will happen when running “mono program regex”, “mono program NFA”, and mono program “DFA”. Each of which will run a hand-picked examples testing regular expression functionality, removing NFA non-determinism functionality, and DFA minimization functionality respectively.

You have the option of adding a regular expression pattern after “mono program regex”, which will run a test on that regular expression. Additional parameters to that will be a series of strings to test the generated machine with. A sample command may look like “mono program regex \”(a\*)(b\*)c\” ab”. Notice that in the regular expression, the Kleene star operations are surrounded by parenthesis. This is to make sure that only the single character is captured. This is because my grammar is defined in such a way that anything to the left of a star or plus is encompassed within that machine. For example, “(ab)+” would be the equivalent of “ab+” in my program.

You have the option of specifying a file containing a DFA after “mono program DFA” in order to run the DFA minimization portion of the program. DFA demo graphs are located under *demos/DFA*. Additional parameters may be appended to the command to specify strings to run the machine with. An example call may look like “mono program DFA demos/DFA/HMU\ 4\_4\_1.txt 01 10 100”.

Similar to the DFA minimization, the algorithm removing NFA non-determinism can be done by running “mono program NFA”. A demo NFA file can be appended to it with additional parameters for strings to run the machine with. NFA demos are located under *demos/NFA*.

**REFERENCES**

[1] Thomas A. Sudkamp. 2005. *Languages and Machines: An Introduction to the Theory of Computer Science (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Robert D. Cameron. (Dec. 1, 1999). *Perl From Regular Expressions in Prolog*. Retrieved from: http://www.cs.sfu.ca/~cameron/Teaching/384/99-3/regexp-plg.html

[3] Wikipedia contributors. (2018, January 10). Thompson's construction. In Wikipedia, The Free Encyclopedia. Retrieved 18:30, April 21, 2018, from https://en.wikipedia.org/w/index.php?title=Thompson%27s\_construction&oldid=819561864