Ryan Darras PJ01 project writeup

Preface

This project was developed using Visual Studio 2017 community edition in C#. There are no customizations made that would interfere with grading or someone else opening and running the project. By default, when the program is run it saves the files into this location ~code\FiniteAutomatonSimulator\FiniteAutomatonSimulator\bin\Debug\Outputs however I have copied this output into the results folder at the top of the zip file. Similarly, that same location except instead of \Outputs, do \Machines in order to read in the machine files. It is currently hardcoded to read in m00.fa, m01.fa, etc so if you don't have all 30 in the Machines folder it won't work. The projects starting file is program.cs. I have commented most of the code in the solution to where it shouldn't be too difficult to understand the strategy as you read along. It should also be noted that I am a graduate student so I was expected to handle both DFA's and NFA's.

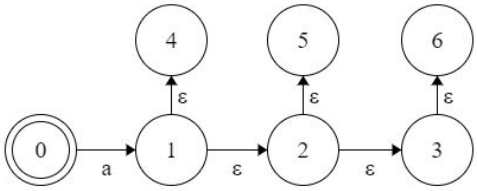
Abstract

This section of this document will explain the basic details of what is to follow, whereas the more detailed descriptions of each element of this project will be described later. For a DFA, I only ever need 1 instance of the machine and I simply take each input string and react based on the machines current state. However, when I add non-determinism into the mix, I needed to become more creative. To do this, I created a list of machines, which are all direct copies of each other, except for what state they are in. An example of this would be that state 0 has two transitions when the input character is 'a', so I would create two machines that follow those two transitions, and delete the current machine. This was incredibly easy, up until the point where I started adding in epsilon transitions. In my previous example, when I add the two new machines to my list, I also need to check for epsilon transitions from their current state and add those copies of the machine to the list with the appropriate current state set. The good news, is that once I get to the end of the string, if I land on any accept state in any of the machines I can simply stop the simulation and call it a success.

Project

NonDeterministicFiniteAutomaton: We defined an NFA as $M = (Q, \Sigma, \delta, S_0, F)$ where Q is a set of states, Σ is the alphabet, δ is a set of transitions, S_0 is the start state, and F is a set of accept states. This data structure contains all of this information, as well as an integer that holds what state we are currently in. When I say "what state we are currently in" I am referring to the general rule that NFA's and DFA's only really know what state they are in at any given time. They do not know what has currently been read

from the input string, or what is remaining on the input string. This data structure has a public “Advance(inputChar)” method that called when iterating on the list of machines that will simply take this machine forward based on the input character. In the case that this input character takes this machine to multiple different states, then it will return all new machines that are possibilities after the input character is processed which are then added to the list to be run on the next input character. I use some tricky recursive strategy to handle all of the epsilon transitions. Whenever I add a new machine to the list, I first check to see if it has any epsilon transitions, which in the case that it does I then need to check the new machine to see if it also has epsilon transitions, so on and so forth. The diagram to the right can be used to explain in more detail. If I have a single machine in my list that is currently on state 0, and my next input is ‘a’, then our list of machines would consist of 6 machines in states 1, 2, 3, 4, 5, and 6. I’m not sure if a situation like this occurred in his 30 machines, but if it did, I handled it. I am completely aware that it isn’t exactly the most optimal solution to completely recreate the machines every single time we go through an input, and that I should instead use one of the existing machines for one of the transitions. However, this way was both easier to develop myself, and it should also be a bit easier to understand from the graders perspective. Built into this class are methods that return information used to create the log files such as “GetAlphabetInOrder()” which sorts the alphabet list, “GetMachineType()” which parses through the variables in this data structure (Q, Σ, δ, S_0, and F) to find if there are any instances that would flag this machine as invalid, or as an NFA over a DFA. To be invalid, our machine needs to have a state numbered outside of 0 - 255, or to contain any element in its alphabet that isn’t a printable character. To decide if the machine is an NFA or not, we need to see if it has any transitions where the input character is ‘`’ because this is what we are artificially using as our epsilon character, and it also checks if any single state has transitions to two other states using the same input character. Note: m11 has two “0,0,16” transitions, which causes my project to flag it as an NFA. I emailed Dr. Bahn and he exclaimed that this is acceptable. If this machine isn’t flagged as an NFA or invalid, then it is passed through as a DFA. I’ve also added many helper methods for this class that makes it a bit easier to read from a perspective that didn’t write it. Instead of seeing crazy long lambda expressions, you see a method name that is descriptive of what the lambda expression should be doing.



Transition: This data structure is a simple struct that holds data. The transition structure holds the initial state, the input char, and the state after input. Each

NonDeterministicFiniteAutomaton has a list of Transitions which represents δ in our definition of a Finite Automaton.

FiniteAutomatonManager: This is a singleton that holds everything together, and knows more about what is currently going on than the machines themselves, since the machines only know what state they are currently in. This class holds the list of machines, the input string, and the collection of accepted strings thus far for this particular machine. When you load up a new machine into the FiniteAutomatonManager, it replaces all the data of the previous machine. We call "CreateFiniteAutomataFromStrings()" which will read in the data provided by the machine files (.fa files) and build our NonDeterministicFiniteAutomaton class as described above. It handles everything from making sure we only create one of each state, I only have one of each element of the alphabet in our alphabet, and so on. It then starts the process by adding the initial machine that is set to the start state to our list. I then call "TestFiniteAutomataAgainstStrings()" which will run every string in the list he gave us through our machine, resetting it to default after each. By "reset it to default" I simply cleared our list of all remaining elements (if there are any) and added a clean machine to the list that is set to the start state. After the string has finished running its course, we check to see if any machines in the list are currently sitting on an accept state. If such a machine exists, then we accept this string and add it to a list of accepted strings.

Program: This is the entry point of this project. The rest of the code was designed to be as stand alone as possible, and all of the testing/debugging/variables/etc can be modified and toyed with here. In the submitted version, it simply iterates 0-30 and calls the FiniteAutomatonManager for CreateFiniteAutomataFromStrings and TestFiniteAutomataAgainstStrings, and prints the results afterwards into the log file and the accepted strings file. The commented code below was used for testing.

Conclusion

All in all, I am happy with the results demonstrated by my simulation. I've tested many corner cases, but as per usual it is likely that I haven't checked everything. If for whatever reason you have any questions/comments I am regularly available via email at ryandarras@gmail.com.