CISS 362: Automata Library Project

(December 10, 2022)

Contents

Chapter 1

Chapter 1: DFAs

This chapter lays out how to use the DFA section of this library. I will also go in depth into the development of this segment of the library.

1.1 Section 1: Development of the DFA library

This segment of the library was developed in stages. For the first stage, I focused on making the DFA work with just strings. I did not establish templeting yet.

For this, our DFA definition is as follows:

S - This is a vector of strings. This is our language, known as our Sigma

Q - This is a vector of strings. This is our collection of states.

q0 - This is a string. This is our starting state.

F - This is an unordered set of strings. This is our collection of states that we accept.

delta - This is an unordered map of a pair of string and string and also a string. This is our transition function.

So we are now able to define our DFA using our DFA definition. Now, as we can define DFAs, we need a way to evaluate strings.

This is where our evaluate function of our DFA comes in. The evaluate function takes in a string and returns a bool. How this function works is that it will take the next character of the string, then make a pair with the current state, then lookup our new state in the delta function. This new state is then stored in the state of the DFA. When we have reached the end of our input, we check to see if it is in our F. This is then returned.

Now that we have our evaluate function, we want to establish a way to call our object like a function. I then overloaded the () operator, calling the evaluate function on the string input. This allows us to evaluate a string without calling our evaluate function outright.

1.2 Section 2: Special Operators on DFAs

There are a couple of operations that you can use in this library. I will go in depth describing the use of these operations and the algorithm used in these operations.

1.2.1 The Complement Operator

One operator that one can use in this library is the Complement of a DFA. This operator is called via a method within the DFA class (called complement). It does not take any inputs and returns a DFA.

The algorithm of this method is really simple. It takes every state in the current DFA and checks if it is present in the final state set. If it is not present, it adds it to a new set. This guarantees that the new final states are the opposite of the current final accept states. Then, this method creates a new DFA object, using the same sigma, states, start state, and delta. However, the final state set is now the new final states set created by the method. It then returns this new DFA.

1.2.2 The DFA Intersection Operator

Another important operator that is included in this library is the DFA intersection operator. For this library, I created a method named intersection() that will take in another DFA and returns a DFA that is the intersection between the caller DFA and the DFA passed in. I will go into the development of this feature in this section, as well as explaining how the algorithm works.

For this method, I first retrieved the collection of states and final states for each DFA. Next I copy the DFA delta function from the passed in DFA. Finally, since the intersection uses tuples to evaluate, I create a new state holder and a new accept state set.

After doing this, I am able to start generating names for new states. I create a hashtable for storing the names, mapping a pair of state names to one single name.

Now that I have all state names generated, I am able to then generate my delta funtion. As we need a tuple for our evaluated state, we evaluate the delta on both the DFA and the passed in DFA for each item in sigma. Next, we fetch the name from our hash table using the current sets and the states that they evaluate to. We then map them to eachother on the value of sigma.

Finally, while we are in our delta generation function, we check to see if the left of the tuple is in F and the right of the tuple is in F'. If they are, we add our evaluated tuple name to the new accpeted set.

Now, we return a DFA generated using the original sigma, the new Q, the new start state (Q[0]), the new F, and the new Delta.

1.3 Section 3: DFA Minimization

To Do

1.4 Section : Translating DFA class to a Template class

To Do

Chapter 2

Chapter 2: NFAs

This chapter lays out how to use the NFA section of this library. I will also go in depth into the development of this segment of the library.

There is a lot of similarities between DFA production and NFA production.

2.1 Section 1: Development of the NFA library

Similar to the DFA, I developed the NFA in stages. As I believed that it would be easier to generate an NFA class before making it a template, my first iteration of the NFA class was hard-coded to accept certain types. Below, I will outline how the NFA class works.

For this, our NFA definition is as follows:

S - This is a vector of strings. This is our language, known as our Sigma

Q - This is a vector of strings. This is our collection of states.

q0 - This is a string. This is our starting state.

F - This is an unordered set of strings. This is our collection of states that we accept.

delta - This is an unordered map of a pair of string and string and also a vector of strings. This is our transition function. You will notice that this is now different than the delta of our DFA class. Because NFAs return a set of states after processing, I wanted to simulate this. This, however, has a drawback of inputing every possible state that you can reach on a singular input when defining the NFA.

So we are now able to define our NFA using our NFA definition. Now, as we can define NFAs, we need a way to evaluate strings.

Like our DFA, our NFA has two methods of evaluating strings. First, we have a method of our NFA class that evaluates a string. Second, we can call our NFA object like a method through the overiding of the () operator.

Our evaluation function is slighly different, however. For evaluating an NFA, what our method does is it takes a character. Then, it makes a pair for every state in our current state field in our object. This is different than our DFA, as it is now a vector of states. From here, we can create a union of states to evaluate. However, NFAs have something else to consider, epsilon transitions.

For epsilon transitions, the object creates a key with our state and how epsilon is represented in the class, "". Then, when checking, after checking each state's character transition, our NFA will then append epsilon transitions to check.

This now becomes our current state in our class. Finally, when we are done proccessing our string, we check every state in our current state to see if it is pressent in the final state set. If it is, the string is accepted.

2.2 Section 2: Creating an NFA from a DFA

Right after making the NFA class, I realized that I wanted to create NFAs using DFAs. This is not just a simple translation, as the delta function for NFAs and DFAs are different. Nevertheless, I was able to come up with a simple and efficient algorithm for delta function translation.

First, I want run over a double for loop over the sigma and state containers. This ensures that my delta has a complete set of keys. From here, I am able to then get the state that the DFA delta function matches to as a string. From here, I add that to an evaluation vector. As DFAs only map to one and only one state per character, I then add the evaluation vector to our new delta function using our key.

Finally, once the delta function is done, I then just assign every object field to the fields in the passed in DFA, with the exception of our new NFA delta function.

2.3 Section 3: Special Operators on NFAs

There are a few special operaters that can be performed on the NFA class via methods. In this section, I will go into the development and application of these special operators.

2.3.1 The Kleene Star Operator

The first operator that I developed for the NFA library was the Kleene Star operator.

The Kleene Star operator is called through the kleene_star method and returns an NFA that has the language of (L(M))*. This means it detects if the language of M is repeated.

For the algorithm, I knew that I would need to establish a new delta function, building off the NFA's current delta. For this, I establish a new start state within the funtion. From here, I copy the states and final states into new containers. These will be used when generating the Kleene Star NFA.

Next, I copy the current delta function into a blank unordered map. From here, I map our new start state to the NFA's old start state using an epsilon transition. Finally, I create transitions from each state present in the accepted states collection for the old NFA to the old start state using an epsilon transition.

The final part of this algorithm is adding our new start state to the accepted states, as empty strings are accepted under the Kleene Star operator.

Finally, the method will build and return an NFA with the same sigma as the original NFA, the updated states with the new start state included, the new start state passed in as the start state, the updated accepted states vector, and the new delta function.

2.3.2 The NFA Union Operator

Another operator that can be preformed on the NFA class is the union of two NFAs. For this, I implemented a NFA Union method called NFA_union(). This method takes in another NFA and returns an NFA that is the Union of the two.

The union of two NFAs determines if a string satisfies the language of at least one of the two NFAs. For this, I took a similar approach to my Kleene Star operator development. I created a new start state. Then from here, I transferred both the Q of our original NFA and the Q of the passed in NFA into two separate vectors. The same can be said for the final states and the delta functions.

After creating these storage variables, I create an unordered map that I will use to create new state names for the passed in NFA states. I map the old name to the new generated name. This is used when I map the delta funtion.

After we generate the names of our states, we can now go and create our delta function. First, I copy over the old delta function onto our new delta function. This removes the redundant looping in my code over our current NFA. Next, I map our new start state to a vector containing the start states of our original NFA and the NFA that we passed in, with our transition being epsilon. Finally, I loop over all the states in the Q of our passed in NFA. Then, I loop over that passed in NFA's sigma. From there, I create transition functions using their new state names in the union NFA.

After I create the delta funtion, I do a union operator on the two NFAs' sigmas. I also do the same for the collection of accepted states.

Finally, I generate and return an NFA using our new sigma, our new Q, our new start state, our new F, and our new Delta.

2.3.3 The NFA Concatenation Operator

Similar to the Union operator, we want to be able to concatenate two NFAs. To do this, we want to add a transition from each accepting state in our frist NFA to the start state of the second NFA on epsilon. Then, we want to ensure that only the accept states of the second NFA are the accepted states.

This is where the NFA_concat() method comes in. It takes in another NFA and returns an NFA that is the concatenation of the original NFA and the passed in NFA.

The development of this method was very similar to that of the NFA_union method. I first created

2.4 Section 4: Converting an NFA to a DFA

Often with NFAs, we want to be able to convert them to a DFA as they compute faster than NFAs. With our library, we will be able to do this conversion with the ”convert_to_DFA()” method in the NFA class. This method does not take in any arguments, but returns a DFA object.

I will go into how the algorithm works for this method now. First, I create a vector to store the new states in, called Q. Then, I create a set to store the accept states in, known as F. After this, I create an unordered_map that will be our new delta function. Finally, I create a vector that stores a vector of strings and another unordered_map. The vector is our ”states” that we need to process and the unordered_map is a table that stores the processed ”state” and the new name of it in the DFA.

Now that we have all our storage variables created, we can get on with computing our DFA from the NFA. We push back our NFA start state into our to_process vector. Then, while the to_process vector has at least one element in it, we pop the top vector off and store it in a vector of states named current. Then, we generate a state name for it and create a vector of strings that is the result of evaluation.

Next, we take all possible epsilon transitions and add them to the eval vector. Then, we take each item in sigma and check to see if there is a transition, adding them to our evaluation vector. If there is not a transition on an element of sigma, we create a state that will be a trap state. This means that the state maps to itself on every element of sigma. Then, we create the missing transition, mapping it to the trap state.

When creating states, we check each item in the eval function. If it is in our NFA’s accept state, we add the new name to the F of our created DFA. DFA state names are created and added to Q upon generation. After creation, if they aren’t in our name table, they get added to the name table and to the to_process vector.

After the algorithm is done, we return our created DFA.

2.5 Section n: Translating to a Template Class

To Do