CSC 173 Project 1
Maryfrances Umeora
Collaborators: Carolina Lion He

Program Files Used

- main
- dfa.c
- dfa.h
- nfa.c
- nfa.h
- Teacher Provided: IntHashSet.c, IntHashSet.h, LinkedList.c, LinkedList.h

How To Run My Project (PLEASE READ)

IMPORTANT: I write my code on the NetBeans IDE, and the Netbeans-generated makefile (called `Makefile-Netbeans`) is very complicated and I'm not sure how to build or run it. Also, although my program is written in the c99 dialect and I turned on the setting to convert warnings to errors, it doesn't show `-std=c99` or `-Wall -Werror` in the makefile. Thus, just in case the Netbeans-generated makefile doesn't work, I used my friend Sifan's computer to do make one (called `makefile`) that more matches what I think the professor does. If the ownership of any of the files show his name rather than mine, that's what it is. I promise everything I've written is mine and mine alone.

Thus, if you are running my code on an IDE, specifically the Netbeans IDE, please remove makefile and rename `Makefile - Netbeans` to just `makefile`. Once you have all the required files, all you have to do is click run `auto.c` (or Build then Run) and you will be prompted by the standard output on what to do next -- it's quite self explanatory.

If you are running on terminal, cd into the project direction, type make and the code will be compiled. To run the code, type `./auto`.

Part I

In this part of the code, I create five DFA objects, set their transitions and accepted states, then allow the user to test their input against the DFA until they enter "quit." The steps I used to create this process are enumerated as follows

1. **DFA**

   In the header file, `dfa.h`, I create a struct which will represent a Deterministic Finite Automata. It has a number of states, an int representing the number of states stored in its accepting array, an accepting IntHashSet, and a double pointer array representing the transition table.

In `dfa.c`, I create a function called `new_DFA` that takes in the number of states and a pointer to an instance of the dfa struct can be made.

2. **Main**
   In main, I create an instance of the dfa struct and fill the accepting array with the relevant state numbers.
   I then fill the transition table with -1s to represent null then I set the transitions with my `set_transition` function. It simply takes in the dfa, the source state, a symbol *x* and the destination that would be reached from that source on that *x*, and adds that destination to to the transition table under row *x* in source src. I also have a `set_transition_all` function which acts as lambda.
   After all this is done, I call my special method called `DFA_Tester`.
3. **DFA_Tester**
   This function, located in my main source file, takes in a message (which is simply the information that will be printed describing the dfa) and the dfa I want to test.
   The first visible thing that this function does is print the informative message, then ask the user for an input ie the string they want to test against the dfa. Next begins an infinite loop which will only break should the user enter "quit." If the user doesn't enter "quit" right away, `DFA_execute` is called and its value is stored in myBool.
4. **DFA_execute**
   This is where all the hard stuff happens. It is located in `dfa.c`.
   First we have an integer named current that represents the current state. Next we loop through the input entered by the user. Another integer which I simply named `a` will get the transition stored in the dfa's transition table at source current on the current letter of the input we are on. As long as there is something there (ie, the number stored there is not -1 which, if you remember, represents NULL), it becomes the new current.
   This will happen over and over until we have looped through every letter in the input string.
   Now we loop through the accepting set. If the current is *in* this accepting array, `DFA_execute` will return 1, which represents true. If not, 0 (false) will be returned.
5. **DFA_Tester again**
   Back to our `DFA_Tester`, the number returned by `DFA_execute` is stored in an integer I named myBool. If myBool is 1, I print true to the user. If myBool is 0, I print false.
6. This process happens five times for each of the five DFAs, And that's basically it.

Part II

In this part of the code, I create four NFA objects, set their transitions and accepted states, then allow the user to test their input against the NFA until they enter "quit." The steps I used to create this process are pretty much exactly the same as with the DFA.

1. **NFA**
   I create a struct which will represent a Nondeterministic Finite Automata. It has a number of states and a double pointer array representing the transition table, an array of accepting states, and a variable representing the size of the accepting array.
   In nfa.c, I create a function called `new_DFA` that takes in the number of states and a pointer to an instance of the dfa struct can be made.
2. **Main**
   I create a few objects of the NFA. I immediately set the accepting set, then add transitions.
   Adding transitions is pretty simple, but for the third one, note that I had to find a way to perform a "Λ-char" operation. To do that, I first set *all* the transitions in the specified source. Then I go into the specific column for the letter and replace the hash set there with an empty hash set.
3. **NFA_Tester**
   This works just like DFA tester. It basically just keeps looping until the user enters quit.
4. **NFA_execute**
   This also works like DFA tester except that instead of having a simple current state, we have a set of possible current states. For each state in that set, we get the transitions there. After everything, we iterate through the final current set and if the accepting state is among the states in that set, we return 1 (true).
5. **NFA_Tester again**
   Back to NFA Tester, the value returned will determine whether "true" or "false" is printed.
6. This process happens four times for each of the four dfas.

Part III

In this part of the code, I convert my first and second NFAs into DFAs. The steps I used to get this are enumerated below.

1. **Main**
   After all my DFAs and NFAs have been, I declare that I will now convert NFA1 into a DFA. I then create a DFA object which is created by a method called convert.
2. **Convert**
   `convert(NFA nfa)` is a function in my nfa.c class.
   The first thing it does is to create a Linked List which represents the `S` column in the subset construction table. I also have a linked list to keep track of checked states.

I then iterate through each of the sub states in `S`, and for each of those, I iterate through the input characters, getting the transitions associated with those characters and sending them into a hashset I call `unionStates`. After that, I check if `unionStates` has already been checked. If it has, I break the loop, but if it hasn't, I increment the value that represents the number of states in the DFA I will be returning, and add unionStates to my S.

Once that is done, I create `dfaTrans`, which will represent the transition table for my returning dfa. Then I iterate through checked states again, this time keeping track of the destination and source states for each state. Once I have this information, I send it into the dfaTable.

The final step is to set the accepting set for our new dfa. I do this by simply iterating through `checkedStates`, and for each item in the original nfa's accepting set, if that item is in checkedStates, then we insert the index of the current state we're on in `checkedStates` into the new dfa's accepting set.

With the number of states, the transition table and the accepting set, we can now call the special constructor for DFAs formed from NFAs.

3. **DFA-from-NFA**

   This is a simple function. It simply returns the dfa with the sent in attributes.

4. **Main**

   Finally, back in main, we send the resulting DFA into the `DFA_tester`. The user can now test any strings against the DFA.

   The same process repeats for NFA2.

5. And we're done!