Matthew Kennedy
CS 5800
Design Document
Due by 4/5/2021

With this document, I will outline possible approaches for different aspects of this project, as well as current research, and the intended design flow.

Possible Approaches
There are 3 different topics I will discuss possible approaches with,
1. UI
2. Input methods
3. Converting DFA into minimized-state DFA

UI
For the UI, there are several UI libraries I am familiar with in the python programming language. The most prominent ones I have used are “tkinter”, and “PyQt”. From the python documentation website, “Tkinter is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)”.

PyQt, on the other hand, is a third-party library, and is the python version of the popular UI library “Qt”. The reason I am choosing PyQt for this project is because the PyQt library provides significant control to the developer, as well as ease of use for blocking calls on the UI thread. This last problem would be an issue in the simulator section of the program - any calls that would block on the UI thread would pose a significant issue for tkinter. This would result in the need of multithreading, which tkinter, in my experience, has not dealt with well. PyQt, on the other hand, deals with this, and other issues, gracefully.

Input Methods
There are 2 input methods for the initial automata I will attempt to implement in this program. The first is manual input - I implemented something similar for the programming assignment that will be integrated into this project. The idea is to not force the user to have an input file for this program, and let them see or adjust tuple elements of the automata as they desire.

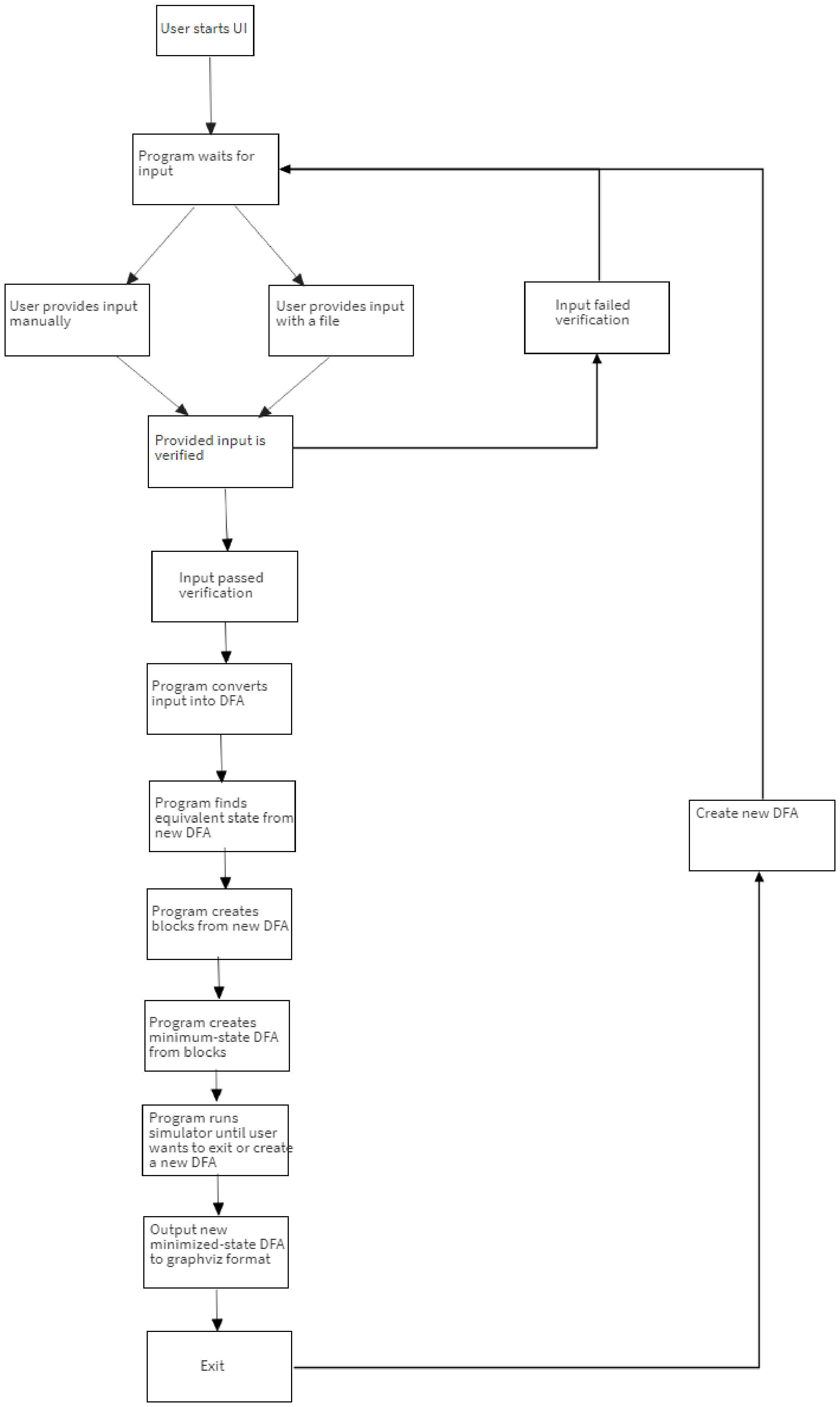
The second method will be having a file input, so the user wouldn’t need to retype all tuple elements every time they’d like to re-run the program. The file input provides the user ease of use, ideally with the click of a button starting the program.

Currently, there are several input fields available to the user, all with helpful hints and error messages to guide the user to properly provide input. An alternative is to provide a file which will populate these same fields, and provide feedback where needed.

Converting DFA into minimized state DFA
The prior assignment that this is based off converts the initial NFA or NFA-ƛ into a DFA. This project will pick up, converting that DFA into a minimized state DFA. This will be done using the handout provided by Professor de Doncker. This conversion will involve 3 steps. The first is identifying all equivalent states. The second step is creating new blocks from these equivalent states. The last step is linking these new blocks as new elements in the minimized-state DFA.

Current Research
Current research has covered a few topics discussed above. Most of the current research comes from the prior assignment done for class. The topics I have already worked on are input methods, input preprocessing, converting an NFA or NFA-ƛ into a DFA, and simulating test strings on a given automaton. From my assignment, I am planning on integrating the work done there, as it is either indirectly or directly relevant and helpful for the continued progress of this project. Next research steps involve creating the initial GUI, as well as implementing the DFA minimization algorithm in steps.

Intended Design Flow
The intended design flow of this project will start with the UI. The user will be asked to input a starting automaton, either manually or with a file. After the user has provided input, the program will then verify the input. If the input fails verification, then the UI asks for input again. If the input passes, a new DFA is constructed from the input. From there, the program will take this new DFA and identify equivalent states. After identification of equivalent states, the program will create blocks from these states, and then fill these into a minimized-state DFA. Lastly, the program will output this minimized-state DFA as a graphviz-file format. The image below outlines the intended design flow.


User starts UI
Program waits for input
User provides input manually
User provides input with a file
Input failed verification
Provided input is verified
Input passed verification
Program converts input into DFA
Program finds equivalent state from new DFA
Program creates blocks from new DFA
Program creates minimum-state DFA from blocks
Program runs simulator until user wants to exit or create a new DFA
Output new minimized-state DFA to graphviz format
Exit
Create new DFA

Overview of Design
From the above picture regarding preliminary design, the design holds true to this. Currently, the program will wait for the user to properly enter input, then when this input has passed all tests, the user can convert an NFA into a DFA, then a DFA in a minimized state DFA, with the click of a button. From there, a new tab becomes available to test strings on the new minimized DFA, as well as view the tuple elements of this minimized DFA.

For new algorithms added to convert a DFA into a minimized state DFA, there are currently 2 steps:

1. Create blocks of equivalent states until all equivalent states are found
2. Create DFA from blocks

Step 1 is a combination of steps from the handout we were provided during the semester - it both A) identifies equivalent states, and B) creates blocks out of the states. The algorithm starts with 2 blocks: all final states, and all non-final states. The algorithm will loop through every block, and compare the blocks in the state for equivalency. If they are equivalent, do nothing. If the states are not, however, remove the second state compared into a new block. Move all states not equivalent into a new block. Once the algorithm passes through all blocks without making a new block for non-equivalent states, the blocks are finished.

Equivalency between states was something that had a difficult-to-implement algorithm - for states to be equivalent, they had to be indistinguishable for each transition they made (either both at non-final states, or both at final states). However, this could be difficult - it could take several transitions for states to become distinguishable. To prove equivalency, there are 100 random strings of length 100, created of random inputs from Σ, to verify equivalency. This provides confidence states are equivalent if they are indistinguishable at every transition.

Step 2 is simpler in concept - first, we find all blocks that contain final states of the original DFA. After this, we go through each block and verify where the states in the block transition for each element of Σ provided. This step verifies states of the same block transition to the same block.

After the new DFA is created, the tuple elements are passed back into the GUI for the user to see. From here, the user can provide strings to process in this to verify if strings pass or fail with the minimized DFA. Additionally, the graphviz file format of this DFA is provided for the user to copy, or save to a file.