1. **Member of the team:**

- Name: Tram Trinh
- NetID: htrinh

2. **Total time spent on the project:**

- Program 1: ~4 hours
- Program 2: ~5 hours
- Total: ~9 hours

3. **Code development: Program 1 - trace-htrinh.py**

a. Planning

- Review the lecture about NFA.
- Understand the input and output format for the trace program.
- Create a function to read the CSV input file and store all the needed data. Create a dictionary to store given transitions as the adjacency list.
- Brainstorm the algorithm to trace all the paths.
- Implement a DFA function that returns the number of possible paths and the number of accepting paths.
- Create a list of strings for each of the given NFA test files to test the correctness of the program. Debug when necessary.

b. Overview of the program

The trace-htrinh.py program will ask the user to input the name of the NFA file. After reading the NFA machines, it will continue to ask the user to input the string that the user wants to test.

The program will print to screen the number of total possible paths and the number of accepting paths. The program also returns the CSV output file that contains the information described below.

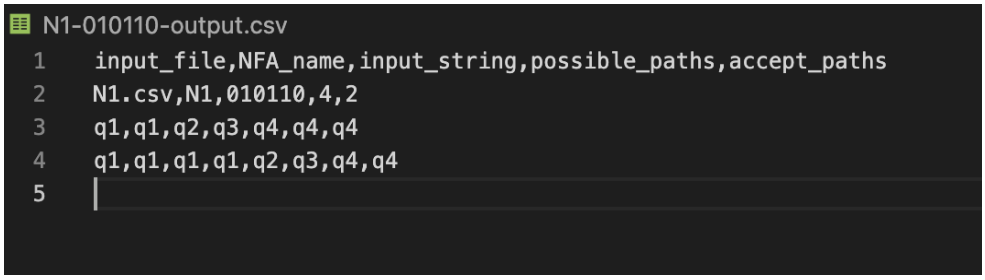
For example, given the NFA file N1.csv and the input string 010110:

```
(base) thbt175@trams-mbp project-02 % python3 trace_htrinh.py
Enter NFA file name: N1.csv
Reading NFA file...: 13it [00:00, 46052.32it/s]
Input a string: 010110
4 2
```

c. Output format

- Name of the output file: [Name of the machine]-[Input string]-output.csv
- The output file contains 2 lines:
    + The first line is the headers: input_file, NFA_name, input_string, possible_paths, and accept_paths.
    + The second line contains all the data associated with each header.
    + Each line after that contains a sequence of states of an accepted path.
- Data in the output file:
    + input_file is the name of the input NFA file
    + NFA_name is the name of the NFA machine
    + input_string is the string that the user input to the program

    + possible_paths is the number of all the paths of the NFA computations on the given string
    + accept_paths is the number of paths that end at the accepting state
- For example, N1-010110-output.csv



4. Language and libraries:

- Language: Python
- Libraries: csv, time, tqdm

5. Key data structures:

a. Directed graph - Adjacency list:

I implemented the deterministic and nondeterministic computation of the given NFA on any input string as a directed graph that indicates all the transitions between all the states.

I implemented the adjacency list for the graph using Dictionary in Python. The key is the first state, while the value is the tuple of the input character and the second state, i.e., Transition[first_state] = (input_char, second_state)

b. DFS:

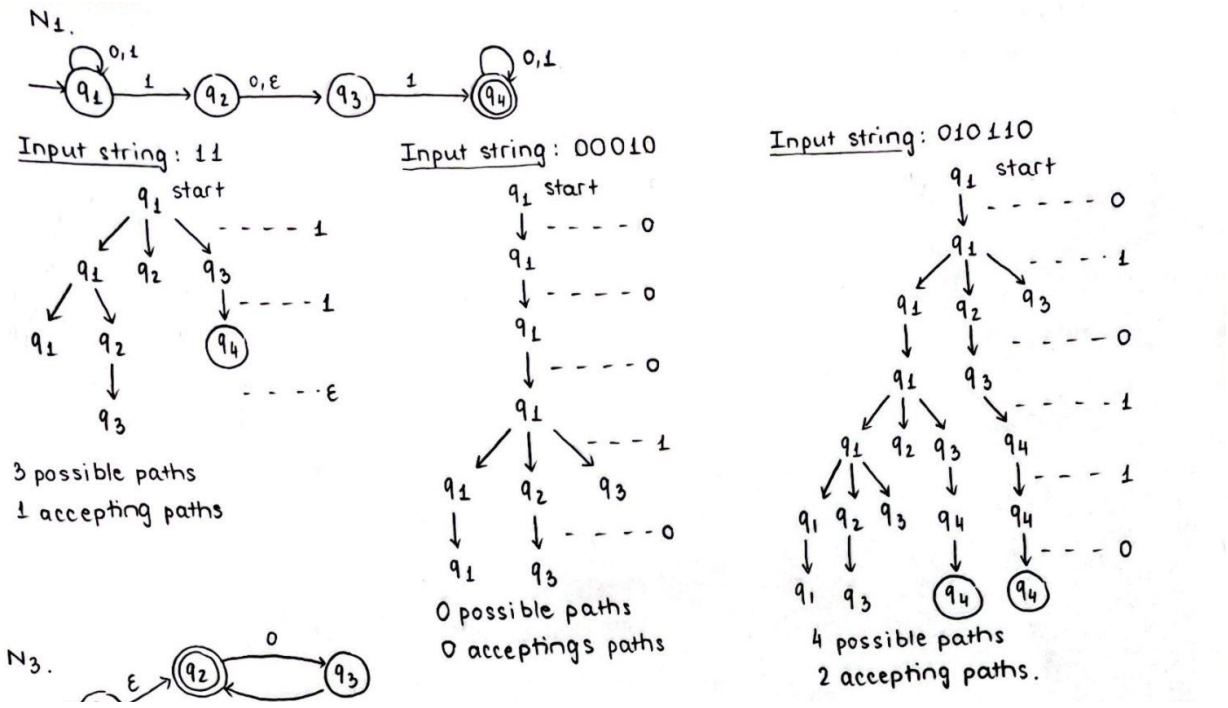
I used DFS to trace all the possible paths of the NFA computation. I have the frontier stack to keep track of the triple (curr, curr_string, epsilon) with curr as the current state, curr_string as the current input string, and epsilon indicating whether the current state accepts epsilon.

I have my DFS tracing all the paths that reach the end of the input string ($), and checking whether the final state at each path is an accepting state. From that, I was able to count all the possible paths of the NFA on the input string and count the accepted paths.

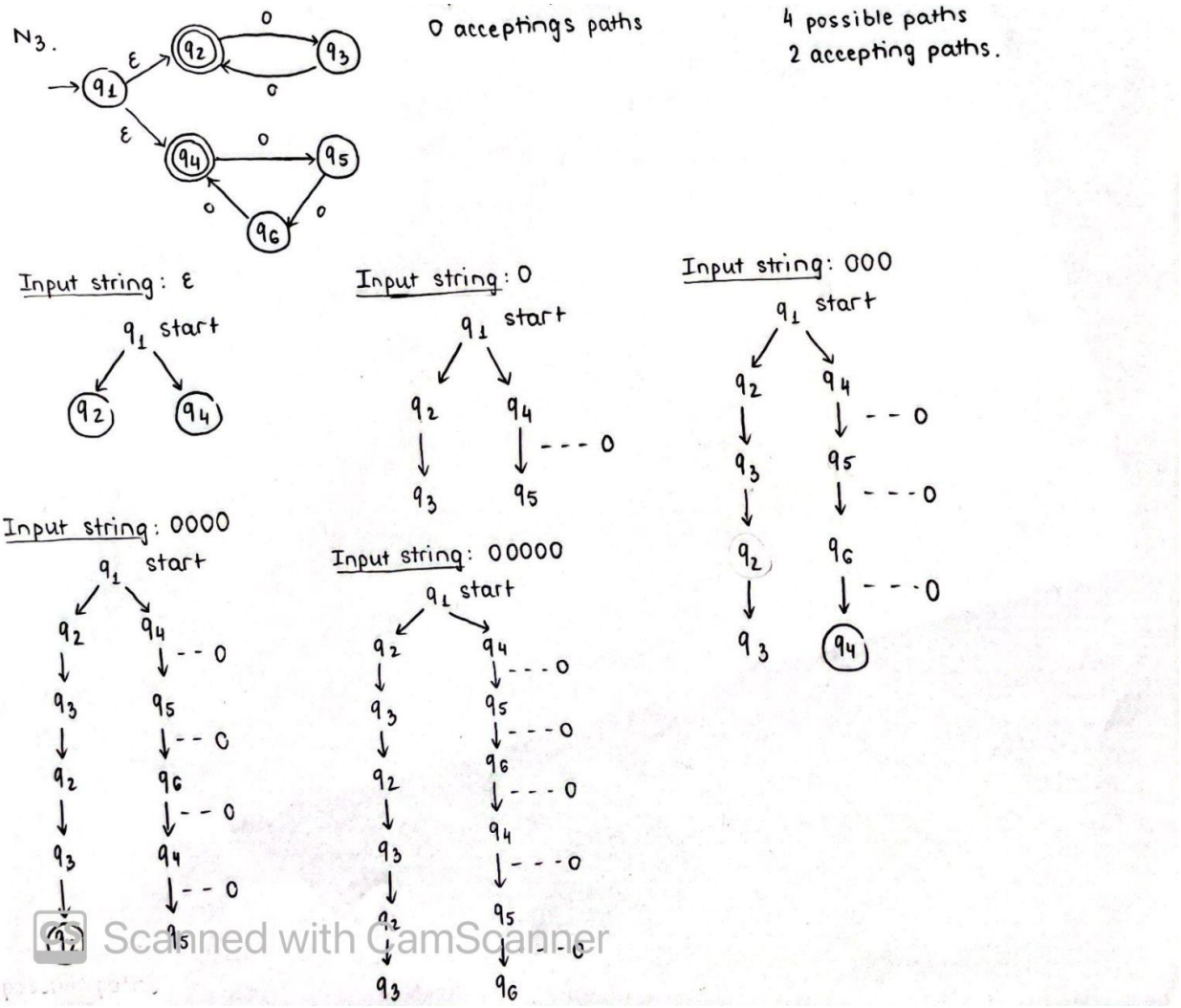
6. Test cases:

a. Testing for N1.csv, N3.csv, and N4.csv:

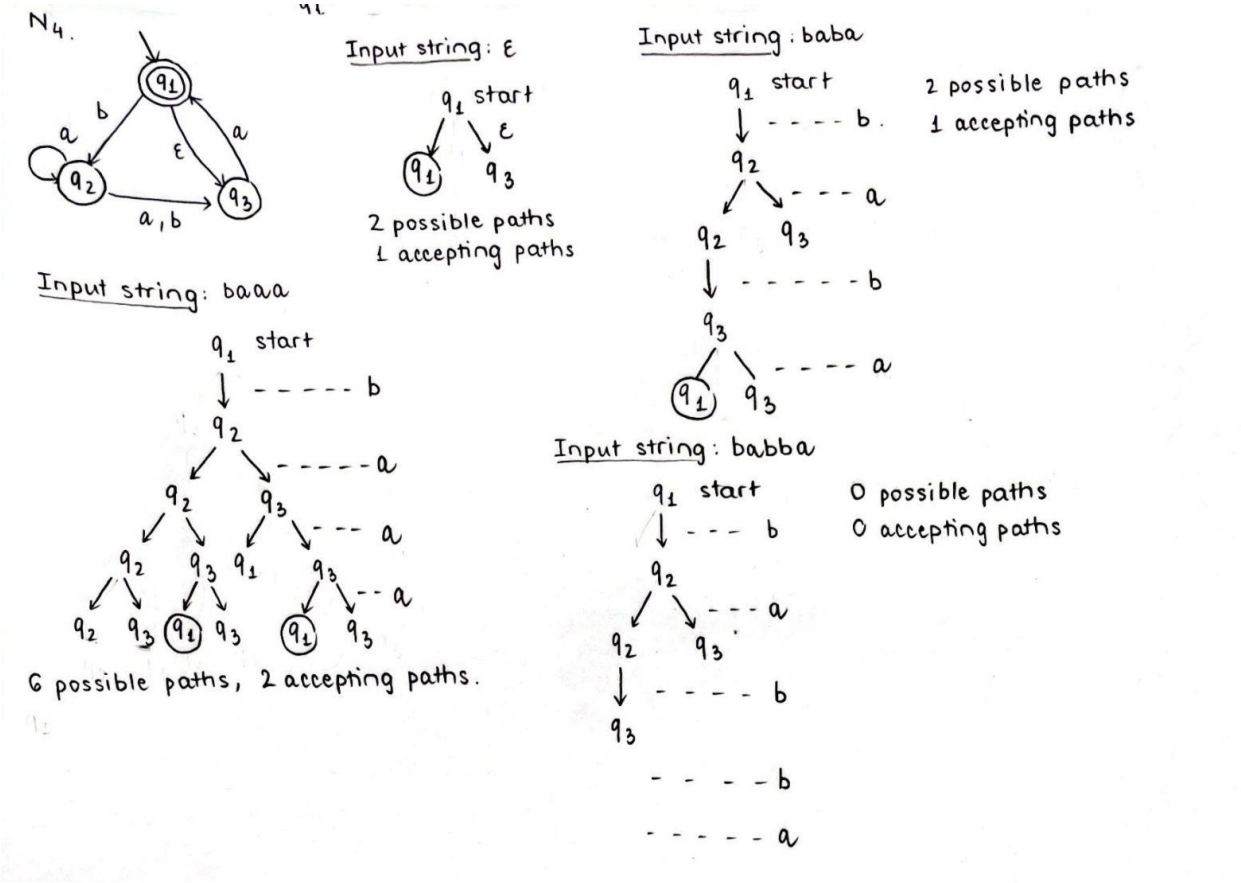
For the three given testing files, I attempted to compile a list of strings for each machine to test the correctness.



For N1.csv, a list of strings that I used are 11, 00010, 010110. The input string 11 makes sure that the trace program includes the epsilon closure in the tree; as illustrated below, one of the unaccepted paths is q1 -> q1 -> q2 -> q3 which includes the epsilon closure at q2. The input string 00010 is the rejected string, so the trace program was able to return 2 possible paths but 0 accepting paths. The input string 010110 is an accepting string with multiple accepting paths, so the trace program returned 4 possible paths with 2 accepting paths.

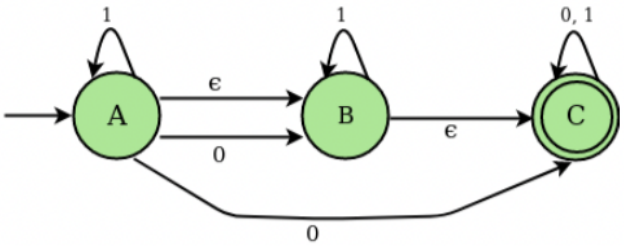


For N3.csv, a list of strings that I used are ~ (empty string), 0, 000, 0000, 00000. The input string ~ makes sure that the program does not throw an error. The trace program returned 2 possible paths (as described below) and none of them is accepting paths. The rest of the input strings are used to check for correctness.



For N4.csv, a list of strings that I used are ~ (empty string), baba, baaa, babba. The input string ~ is used to check if the empty string can be computed by the machine. Since the start state of the N4 machine is also the accepting state, ~ is an accepting string. Therefore, the trace program returns 2 possible paths with 1 accepting path for the empty string. The rest of the input strings are used to check for correctness.

b. Extra NFA machine: Problem 7 on HW4



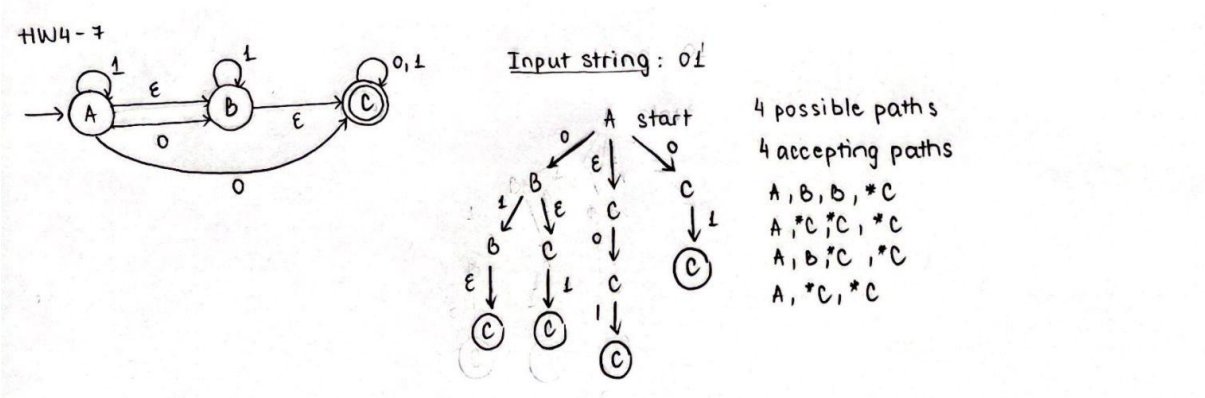
I found this NFA interesting because there is an epsilon from state A to state B and another epsilon from state B to state C.

The input CSV for this NFA is formatted as

HW4-7.csv

```
1   HW4-7
2   A,B,*C
3   0,1
4   A
5   *C
6   A,1,A
7   A,~,B
8   A,0,B
9   A,0,*C
10  B,1,B
11  B,~,*C
12  *C,0,*C
13  *C,1,*C
```

The input string is 01 which is also a question on problem 7 on HW4. The program returns the correct answer which is 4 possible paths with 4 accepting paths.



7. Extra program: Program 2 - nfa2dfa-htrinh.py

a. Planning:

- Review the instructions for converting epsilon-NFA to DFA.
- Understand the input and output format for the nfa2dfa program.
- Create NFA and DFA class objects
- Create a function to read the NFA machine from the CSV input file and store all the needed data in the NFA object.
- Create a method in the NFA class to convert the input transitions to the dictionary in the following format: T[first_state] = (input_char, next_state)
- Create a method in the NFA class to get the epsilon closure. The function takes in a state in the machine and returns the epsilon closure of the given state.
- Create a function to iterate through every single state in the NFA and create a dictionary for epsilon closures of all the states.
- Create a function to convert NFA transitions to DFA transitions.
- Create a function to convert all NFA data to DFA data and store it in the DFA object.
- Create a method inside the DFA object that get all the data for the set of 5-tuples that defines the DFA and writes them to the CSV output in the format described below.

b. Overview of the program:

The nfa2dfa-htrinh.py program will ask the user to enter the NFA file name. It will then convert the given NFA machine to DFA and return the CSV output that contains all the transitions of the DFA in the format described below.

c. Output format:

- Name of the output file: [Name of the machine]-nfa2dfa.csv
- For example, N4-nfa2dfa.csv
- The output file contains multiple lines:
    + Line 1: Name of the machine
    + Line 2: List of all the states for Q
    + Line 3: List of characters from ∑
    + Line 4: The start state
    + Line 5: List of accepting states
    + All transitions lines
- Each transition is defined by a single line that consists of three items separated by commas:
    + Name of a state that the machine might be in
    + A character from ∑
    + Name of a state that the machine may go into if that character was found next on input
- All the states with "*" at the front are accepting states
- "Err" is the trap state.
- The name of each state in the DFA is the concatenation of prior stat names from the NFA and separated by "+".

d. Data structures:

To find the epsilon closure of a state, I used DFA with a frontier stack to get what to evaluate next and a visited dictionary to track which states have been evaluated. The list of keys of the visited dictionary is the epsilon closure of the given state.

I also implemented DFS in the function convert_2dfa to convert NFA transitions to DFA transitions starting from the epsilon closure of the NFA start state which is the start state of the converted DFA. The dfa_transitions is a dictionary that is used to keep track of new states for DFA.

The function returns the list of all DFA states and the dictionary of the DFA transitions. In the case that there exists an "Err" state in the list of the DFA states dfa_states, the program will iterate through all the transitions in the dfa_transitions dictionary to update transitions to the Err states.

e. Test cases:

To test the nfa2dfa program, I compiled my program on three given NFA test files. I used the N4 machine to debug my program.

Below are the given NFA N4 machine in the state diagram and the input CSV file.

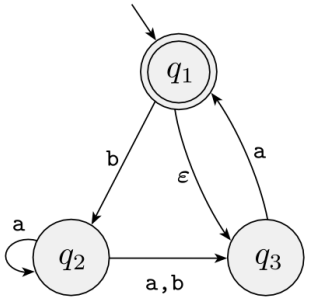


FIGURE 1.36
The NFA N_4

N4.csv

```
N4-fig. 1.36,,
*q1,q2,q3
a,b,
*q1,,
*q1,,
*q1,b,q2
q2,a,q2
q2,b,q3
*q1,~,q3
q3,a,*q1
q2,a,q3
```

The nfa2dfa_htrinh.py took in the NF4.csv as the input and converted the given NFA to an equivalent DFA. It then returned the output file N4-nfa2dfa.csv:

N4-nfa2dfa.csv

```
N4-fig. 1.36
*q1+q3,Err,q2,q2+q3,q3,*q1+q2+q3
a,b
*q1+q3
*q1+q3,*q1+q2+q3
*q1+q3,a,*q1+q3
*q1+q3,b,q2
Err,a,Err
Err,b,Err
q2,a,q2+q3
q2,b,q3
q2+q3,a,*q1+q2+q3
q2+q3,b,q3
q3,a,*q1+q3
q3,b,Err
*q1+q2+q3,a,*q1+q2+q3
*q1+q2+q3,b,q2+q3
```

Below is the reduced DFA that is equivalent to the NFA N4. From the output above, we can see that the nfa2dfa was able to list all the transitions in the equivalent DFA of the NFA N4.

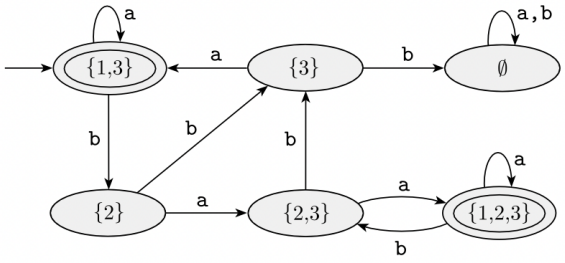
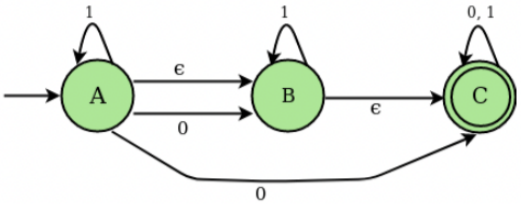


FIGURE 1.44
DFA D after removing unnecessary states

The program also converted the other two NFA N1 and N3 to equivalent DFA in order to test for correctness. The outputs for those two machines are in dropbox as N1-nfa2dfa.csv and N3-nfa2dfa.csv.

I also used the NFA from Problem 7 on HW4 to test the nfa2dfa program. Below is the state diagram of the NFA and the transition table of the equivalent DFA from the HW4 solution.



NFA HW4-7

State	Input=0	Input=1
$\{A, B, C\}$	$\{B, C\}$	$\{A, B, C\}$
$\{B, C\}$	$\{C\}$	$\{B, C\}$
$\{C\}$	$\{C\}$	$\{C\}$

Transition table of the equivalent DFA

The output file for the NFA HW4-7 that the program nfa2dfa returned:

HW4-7-nfa2dfa.csv

```
1   HW4-7
2   *C+A+B,*C,*C+B
3   0,1
4   *C+A+B
5   *C+A+B,*C,*C+B
6   *C+A+B,0,*C+B
7   *C+A+B,1,*C+A+B
8   *C,0,*C
9   *C,1,*C
10  *C+B,0,*C
11  *C+B,1,*C+B
12
```