COS284 Project

University of Pretoria
Department of Computer Science

Online Publication: 25 September 2023

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Plagiarism Policy

- All work submitted must be your own. You may not copy from classmates, textbooks, or other resources unless explicitly allowed.
- You may discuss the problem with classmates, but you may not write or debug code together.
- If you use any advanced material, you must cite these sources.

Restrictions

You're encouraged to utilise any resources available within the confines of assembly and standard C libraries. Although you may create multiple ASM files, C file coding is prohibited. Familiarising yourself with header file usage could be beneficial for this project, but the implementation approach is ultimately your choice. **Take note: Do not alter the provided makefile. If it cannot compile your project, neither will FitchFork**.

Group Work

This project requires you to work in groups of at least 3, and at most 5 members.

Background

The COS210 faculty initiated a project with student developers to design a program for Deterministic Finite Automata (DFA). This program would allow lecturers to define a DFA in a file, which the program could then construct and simulate with input strings.

Regrettably, the initial team of developers did not complete the project and the lecturers are unsure of its accuracy. They have now expressed a preference for assembly language.

Your task is to finalise this project in three stages, with each subsequent stage building on the former. Ensure you begin promptly. Refer to FitchFork for the project's due date.

Deliverable 1

Before being able to do anything fancy, we first need to construct DFA. The DFA will be specified in a file, and your program will read the file to build the DFA. The structures used will be as follows:

```
typedef struct
{
    int id;
    bool isAccepting;
} State;

typedef struct
{
    int from;
    int to;
    char symbol;
} Transition;

typedef struct
{
    State *states;
    Transition *transitions;
    int numStates;
    int numTransitions;
    int startState;
} DFA;
```

The DFA will be specified in a file with the following format:

```
3,6
0,1,2
0
0,1,a
0,1,b
1,2,a
1,2,b
2,0,a
2,0,b
```

- The first line specifies the number of states and transitions, respectively.
- The second line specifies the IDs of the states.

- The third line specifies the IDs of the accepting states. Also comma separated.
- The remaining lines specify the transitions. Each line has the format `from,to,symbol`.
- You may assume the starting state is always `0`.

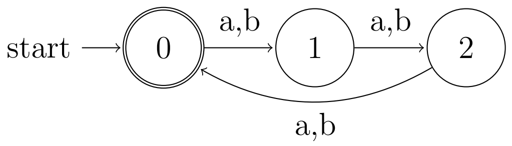
You will need to implement the following interface:

```
DFA* readDfa(const char *filename)
```

This function will read the file and construct the DFA. It will return a pointer to the DFA. If the file does not exist, or the file is not formatted correctly, the function will return `NULL`. An example of how to use this function is as follows:

```
DFA *dfa = readDfa("dfa.txt");
```

Where the DFA using the file specified above would look like this:



Take note that there are 6 transitions in this DFA as previously specified.

Deliverable 2

Now that we can construct DFA, we need to be able to simulate input strings over them. We will do this by implementing the following interface:

```
bool simulateDfa(DFA *dfa, const char *inputString)
```

This function will simulate the input string over the DFA. If the DFA accepts the string, the function will return `true`, otherwise it will return `false`. An example of how to use this function is as follows:

```
DFA *dfa = readDfa("dfa.txt");
bool accepted = simulateDfa(dfa, "ababab");
```

And assuming the DFA is the same as the one specified in the previous deliverable, the output would be `true`. It is up to you to figure out how boolean values work in C and how to use them in assembly. **The input strings will be C strings**.

Deliverable 3

Now that we can construct DFA and simulate input strings over them, we need to be able to check if two DFA represent the same language. We will do this by implementing the following interface:

```
bool sameLanguage(DFA *dfa1, DFA *dfa2)
```

This function will return `true` if the two DFA represent the same language, otherwise it will return `false`. An example of how to use this function is as follows:

```
DFA *dfa1 = readDfa("dfa1.txt");
DFA *dfa2 = readDfa("dfa2.txt");
bool same = sameLanguage(dfa1, dfa2);
```

There are different approaches to verifying this. An example approach will be listed below, however, it is quite a lengthy process and you are free to implement your own approach. The example approach is as follows:

Algorithm to Check if Two DFAs Accept the Same Language

Given two deterministic finite automata (DFAs) A_1 and A_2, we construct a new DFA A_d to determine whether A_1 and A_2 accept the same language.

Formal Definitions

Let A_1 and A_2 have the following components:

- Q_1, Q_2 : The sets of states in A_1 and A_2, respectively.
- Σ: The alphabet.
- δ_1, δ_2: The transition functions, $\delta_1 : Q_1 \times \Sigma \to Q_1$ and $\delta_2 : Q_2 \times \Sigma \to Q_2$.
- F_1, F_2: The sets of accepting states in A_1 and A_2, respectively.

Constructing A_d

The new DFA A_d has the following components:

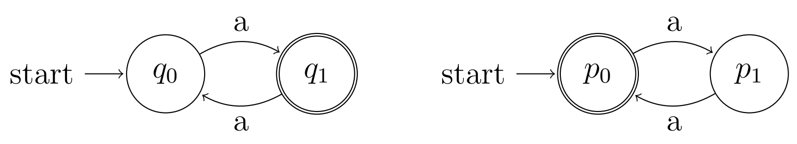
- **States**: $Q_d = Q_1 \times Q_2$
- **Alphabet**: $\Sigma_d = \Sigma$
- **Transition Function**: $\delta_d : Q_d \times \Sigma_d \to Q_d$ defined by

$$\delta_d((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

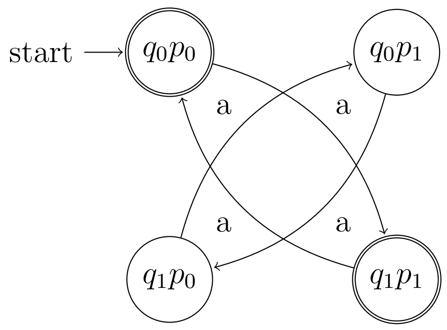
- **Accepting States**: F_d contains all pairs (q_1, q_2) where $q_1 \in F_1$ and $q_2 \notin F_2$, or $q_1 \notin F_1$ and $q_2 \in F_2$.

Algorithm

To determine if A_1 and A_2 accept the same language, we explore A_d from its initial state $(q_{1_{\text{init}}}, q_{2_{\text{init}}})$. If we ever reach an accepting state in A_d, A_1 and A_2 do not recognise the same language. Otherwise, they do. Consider two DFAs, A_1 and A_2, defined as follows:



Combined DFA A_d:



By analysing the reachability from (q_0, p_0), which corresponds to the initial state q_0p_0 in A_d, we can confirm that A_1 and A_2 do not recognise the same language.

Given Files

You have been provided a lot of files to help you get started. You have a received an initial helper function and an "include" file with your structures setup already. Look at how the include pre-processor directive works inside **initDfa** to help keep your code clean.

Mark Distribution

Task	**Marks**
Constructing DFAs	30
Verifying Input Strings	30
Determining Language Equivalence	40
Total	**100**