CSE 30151 Theory of Computing
Spring 2018
Project 2-Finite Automata

Version 1

Contents

1 Overview

The goal of this project is to have each student understand at a deep level the functioning of a **finite automaton** (FA), how "programs" for such a machine should be written, and what we mean by "complexity" of such programs when executed on real inputs. This design should set the stage for later projects on alternative classes of automaton.

This project is designed to give students a relatively wide range of options, both in what is attempted, the platform on which the project is executed, and how teaming with other students may be performed. Each option is designed so that each student will have approximately the same amount of educational experience.

Each team shall adopt a *team name* that is used as part of all program names, and referenced in documentation.

2 Valid Options

2.1 Project Options

The project has a spectrum of different options that may be pursued:

- *dfa*: a **deterministic finite automata** simulator that is capable of accepting a description of a DFA and then executing that machine against a set of strings, reporting back for each whether or not the string was accepted by the machine.
- *fst*: a **finite state transducer** that operates just like a DFA but also outputs a character at each transition that may be used to interact with the outside world. It should be possible to run an FST simulator as a simple DFA also.
- *nfa2dfa*: a program that takes a valid description of an NFA, and creates a valid description of a DFA that accepts exactly the same language. This DFA description shall be executable on a dfa simulator.
- regex2nfa: a program that takes a **regular expression** and creates an NFA that accepts it. The description of an NFA should be compatible with a *nfa2dfa* program, which means that it can be translated into a form that can be run on a DFA simulator.

The *nfa2dfa* and *regex2nfa* are to be compatible with either a DFA or FST simulator.

2.2 Platform Options

There are two platforms on which this project may be run: totally on a conventional computer such as your laptop or the student machines, an Arduino single board embedded computer, and/or a mix of Arduino and conventional platforms.

2.3 Teaming Options

There are also two teaming options possible. First is **collaborative** where the members of the team may freely share development and coding, submit single sets of code and documentation, and receive a common grade. Second is **inter-operable** where each student develops their own code and documentation independently, but may use the programs produced by other students/teams to prove out their code. An example of the latter might be for a student who is interested in development of just a *regex2nfa* module, and uses the *nfa2dfa*

and *dfa* from other teams to prove it out. In such cases, each student/team receives an independent grade based on just their own work and submission.

2.4 Overall Combinations

In a collaborative team, groups of up to three students may work freely together. The number and mix of programs that must be written depends on the number of students, and include the following options:

- Single student: any one of the following:
  - *fst* on a conventional computer
  - *dfa* on an Arduino
  - *nfa2dfa* on a conventional computer (Requires an inter-operable agreement with some other team that has a *dfa*)
  - *regex2nfa* on a conventional computer (Requires an inter-operable agreement with some other team that has a *dfa* and a *nfa2dfa*)

  The rationale for requiring only a simpler DFA rather than an FST when running on an Arduino is to reflect the need for additional programming on both the host and the Arduino side to transfer data from host to Arduino, and back.
- Two person collaborative teams:
  - *fst* on an Arduino, including designing demo examples that interact with the real world, providing inputs from physical sensors and driving physical output devices. The instructor has a wide supply of material for such demonstrations.
  - *dfa* and *nfa2dfa* on a conventional computer. The *nfa2dfa* must drive the *dfa* simulator.
  - *nfa2dfa* and *regex2nfa* on a conventional computer. (Requires an inter-operable agreement with some other team that has a *dfa*)
- a three person collaborative team is expected to develop all three of *fst*, *nfa2dfa*, and *regex2nfa*.

Other combinations may also be possible - see the instructor for permission.

2.5 Programming Options

You are free to use C, C++, Python, or even (ideally) the C environment for your Arduino. On conventional machines, Python in particular has proven in the past as perhaps the most efficient way to get decent working code (the overall goal). If you use Python, feel free to use the *time*, *csv*, *sys*, *copy*, *itertools*, and *pygame* modules. Use of any other modules requires preapproval of the instructor.

Many of the files are defined to be in "'csv" format, a text file format where each line is a series of text strings separated by commas. The Python *csv* library is particularly useful for processing such files. Note that the use of spreadsheets for generating test files is useful, since there is typically a "save as .csv" option. Note also that it may be that such files are produced on a Windows machine that adds a carriage return and a linefeed to the end of each line, versus just a line feed under Linux.

If you use your Arduino, you may use any built-in library. Any other libraries should be passed by the instructor before use. Also the instructor has a few display units and quite a few Arduino-compatible sensors and actuators that can be loaned out, along with documentation on drivers.

2.6 Honor Code Considerations

In any case, all aspects of the ND Honor Code (https://honorcode.nd.edu/the-honor-code/) and CSE-specific interpretation (http://cse.nd.edu/undergraduates/honor-code). Members of a team are free to share code within themselves but not with students outside the team. In an inter-operable arrangement, neither student team can look at or be involved in the development of the other team's code, although reporting bugs is permissible. Also in such an arrangement, failure of one side to complete coding in a timely fashion is not an excuse for the other side. In no case is any code from outside the class, other that that explicitly authorized either here or by the instructor, allowed. In particular, this includes code from prior years' projects that may have been similar.

3 Program Details

Completion of this project will involve two parts. First is developing each required module. Doing this involves both programming and algorithm understanding. Second is developing your own automata, and then using your tools to simulate it and show it does what you expect.

We are not after world-class performance or beautiful code here, but just some relatively simple code that is functional and from which you can draw insight. You are free to go beyond the minimal requirements and produce enhanced code, and if in the instructor's view it truly represents valuable extensions, then it will be considered for extra credit (see Section **??**).

3.1 DFA

This program, to be named **dfa-*team***, where *team* is the name of your team, will simulate a DFA that is programmed to accept/reject strings depending on if they are members of some language. It must take two files at input: an input "machine file" defining the DFA to be simulated (Section 3.1.1), and an input file providing multiple character strings to run one at a time against the machine.

After reading in the machine file, this program should read the second file one line at a time. Each line is a character string represents a separate input string for running on the DFA. Before each string is processed from the second file, the simulated DFA should be reset to its start state, and all statistics should be cleared. The name of the test file shall be echoed to stdout. Then the FA should be allowed to run against the next line of text from the input file. At the end, your program should signal if the string was accepted or not. Section 3.1.2 defines the output that should be generated during the simulation.

In designing your simulator, you should design in an extra "Trap" state to which any combination of state and input not defined by the machine file should go. This trap state is assumed to be part of all designs, and need not be defined in the machine file. It is never in the set of final states. It is there to both catch errors in machine descriptions and to simplify the description of DFAs. A DFA designer is of course permitted to define and use their own trap state.

3.1.1 DFA Machine Format

A file providing the rules for a DFA shall consist of a .csv file where each line provides distinct information about the DFA as follows:

- Line 1: The "name" of the machine being defined. This should be echoed to stdout before the rules are echoed.
- Line 2: Σ - the alphabet to be used as input: only single ASCII letters are allowed, comma separated, as in a,b,c,... Any ASCII character other than $\sim$ or "," is allowed. The $\sim$ character is reserved to stand for ϵ in transition rules for NFAs.
- Line 3: Left blank (here for compatibility with FST designs)
- Line 4: **Q** - the names of the states, separated by commas, as in q0, q1, ... There is no constraint on the length or character set of a state name.
- Line 5: q_0 -The name of the state that should be considered the start state.
- Line 6: **F** - A comma separated list of state names that should be marked as accepting states.
- Line 7 and beyond: δ - one transition rule per line, in a comma-separated format:
  $Initial\_State\_Name, Input\_Symbol, New\_State\_Name$
  The $Input\_Symbol$ is any symbol from the defined Σ from line 2, or (when describing an NFA) optionally a $\sim$ that stands for an ϵ.

This file should be read in by your simulator, and internally processed to whatever representation you are using to represent the transition function. As each rule is read in, an integer "Rule #" should be associated with it, assigned in sequential order. The Rule #, followed by a ":", and an echo of the rule should be dumped to stdout so that later traces that reference the rules can be followed.

An acceptable simulator need not do any "check" that a character used in a transition is in Σ from line 2; if such a character is used in an input string, a branch to Trap is sufficient. Likewise, depending on your internal representation for the transition function, Line 3 (Q) need not be explicitly parsed and used.

Also, if the machine file given to your simulator has multiple transitions from the same state with the same input symbol (i.e. an NFA transition), you may choose any approach as to what happens, and you do not need to error check that such transitions are in the file.

Note also that this format was designed to allow interoperability of DFAs and FSTs (see Section 3.2.1). A *dfa* simulator when given an FST design will skip both line 3 (the FST output alphabet), and should ignore any fourth item on a transition line.

3.1.2 DFA Simulation Output

When the simulator is processing an input line from the second file, it should output to stdout a trace of its execution as follows:

- At the start, the input string being processed, prefixed by "String: "
- For each transition, the following on a separate line:
  $Step\_number, Rule\_Number, Initial\_State\_Name, Input\_Symbol, New\_State\_Name$
- After all transitions have been performed, print either "Accepted" (if the last state was an accepting state) or "Rejected" (otherwise)

The output should be compatible with a ".csv" form, allowing a redirect to a ".csv" file so that the output can be read by a spreadsheet program such as Excel.

If there is a transition not defined in the machine file, you may assume the state to which the transition goes is the builtin "Trap" state.

After the last line of strings in the input file has been processed, there should be a final line output that gives the number of lines processed, the number that were accepted, and the number rejected (rejection includes ending up in the built-in Trap state). This line will be used by the grader to check for correct execution of test cases.

3.1.3 Implementation on Arduino

If your DFA is implemented on Arduino, there are several possible variations. The major one is how the machine file is input into the Arduino and processed by the sketch. Most likely, the file is actually read in by the host computer, and transferred to the Arduino via calls using the *firmata* library. This thus involves writing two programs: the Arduino sketch and some program in the host (this is why a single person team on Arduino need only do a DFA and not an FST).

Within this option there are variations on how the parsing and processing of the file is performed. Doing it on the host means that a variety of libraries, such as Python's *csv* library considerably simplify coding. In contrast, the host side could simply read in the file, and send to the Arduino.

The second major variation involves how the input string is input. Again, a host program could read the file, send a line at a time to the Arduino, and then receive the stream of output text. Alternatively, the Arduino monitor function could be used to type in the string, and the output could go to the monitor (with an on-board LED representing the DFA being in an accepting state.

3.1.4 Instructor-Supplied Test Files

Under the Projects tab of the class website https://www3.nd.edu/ kogge/courses/cse30151-sp18/index.html there is a directory called Project2. Within this directory there are several test files that can be used with **dfa-*team***:

- Four different machine definition in both .txt and .csv formats: M1, M2, M3, Mystery.
- For the first three machines, sets of strings that should be accepted or rejected by the associated machines
- For the Mystery machine, a set of strings whose acceptance is unknown

3.2 FST

An FST is a DFA where on each transition, a symbol from a second alphabet Γ is output. The second option program, to be named **fst-*team***, where *team* is the name of your team, will simulate a FST. As with the DFA, it must take two files at input: an input "machine file" defining the DFA to be simulated, and an input file providing multiple character strings to run one at a time against the machine. In most cases, we would expect the code for this program to be a relatively straight-forward modification of that for the DFA.

3.2.1 FST Machine Format

The format for an FST simulator is very similar to that for the DFA (Section 3.1.1):

- Line 1: The "name" of the machine being defined. This should be echoed to stdout before the rules are echoed.
- Line 2: Σ - the alphabet to be used as input: only single ASCII letters are allowed, comma separated, as in a,b,c,... Any ASCII character other than $\sim$ or "," is allowed. The $\sim$ character is reserved to stand for ϵ in transition rules for NFAs.
- Line 3: Γ - the alphabet to be used as output. Again this is a comma-separated list, but may take up one of two forms (implementer choice):
  - single ASCII letters, comma separated, as in a,b,c,
  - comma-separated list of hexadecimal numbers, as in "0x12". (This is an option particularly for Arduinos where the output may be a set of digital devices where binary codes drive what should happen.)
- Line 4: **Q** - the names of the states, separated by commas, as in q0, q1, ... There is no constraint on the length or character set of a state name.
- Line 5: q_0 -The name of the state that should be considered the start state.
- Line 6: **F** - A comma separated list of state names that should be marked as accepting states.
- Line 7 and beyond: δ - one transition rule per line, in a comma-separated format:
  $Initial\_State\_Name, Input\_Symbol, New\_State\_Name, Output\_Symbol$
  If there is no fourth item in a transition rule, then no output character should be generated.

The $Input\_Symbol$ is from Σ and $Output\_Symbol$ is from Γ.

Note that an FST design file given to a DFA simulator will run correctly except that it will not output any characters.

3.2.2 FST Simulation Output

The output of an FST should be again very similar to that of a DFA (Section 3.1.2), except for each transition the output character needs to be included. Further, at the end of processing a string, the output string should be printed.

3.2.3 Implementation on an Arduino

An Arduino implementation of an FST is again very similar to that of a DFA (Section 3.1.3), with the extra requirement as to how to generate and display the output string. In particular, an Arduino implementation would ideally generate an output that drives discrete outputs to physical devices. In addition, it could be run in a mode where the input symbols are physical sensors, push buttons, etc.

3.3 NFA-to-DFA Translator

This program option, to be called **nfa2dfa-*team***, where *team* is the name of your team, should read in a machine file in the same format as for the DFA (Section 3.1.1), and produce an output file that represents a valid DFA version of the original input. It is expected that this original input file is for an NFA, that is, there may be either multiple transitions from the same state and same input symbol, and/or transitions with ϵ as the character. The output of your translator should be totally compatible with a DFA simulator as discussed ion Section 3.1, and should be demonstrated by running the output with appropriate test

strings on that DFA simulator. The algorithm behind this program should resemble that of the book's Theorem 1.39.

In the same directory as the DFA test files (see Section 3.1.4), there are several NFA files that should be used as test files. After translation, these files should be run through the DFA to demonstrate their correctness.

Also as with the DFA, each student in a collaborative team should generate their own NFA test machine and test strings, and verify that the output is correct.

Given the nature of this program, it may be most efficient to write it to run on your host, and not your Arduino. Certainly, however, if you have an Arduino DFA, the output of this program should be a file that drives it directly.

3.4 Regex to NFA

The final program option, to be named **regex2nfa-*team***, where *team* is the name of your team, takes a string representing a regular expression, and using Lemma 1.55 from the book to construct an NFA. The output of this tool should be compatible with the *nfa2dfa* tool, which in turn should then run on a DFA simulator. All three programs then gives us the capability of writing a regular expression and then getting out a DFA which accepts it.

For test programs, each student in the team shall develop a different regular expression, and verify that the output is correct.

Given the nature of this program, it may be most efficient to write it for your host, and not your Arduino. Certainly, however, if you have an Arduino DFA, the output of this program should be a file that feeds an *nfa2dfa* program which in turn drives the Arduino.

3.5 Student-supplied Test Cases

In addition to instructor-supplied DFA and NFA machines and test program, each individual student shall create their own non-trivial test case for the "highest" program written by their team. For example a one-person team who builds a *dfa* or *fst* must design at least one new DFA or FST. A two-person group that develops a *nfa2dfa* must then have each student develop their own NFA and show that it runs correctly through both programs. For a three-person team, each person must develop a separate non-trivial regex and show that it has been translated correctly through all three tools. These test cases should be included with the main code, and documented in each student's teamwork report (Section 4.2). The name field of each machine should thus include the *netid* of the student doing the design.

4 Documentation

Two pieces of documentation, both in PDF format, are required. One (entitled **readme-*team*.pdf** is submitted once by the entire team in the Sakai directory associated by one of the team members; the other **teamwork-*netid*.pdf** is submitted separately by each member of the team in their own directory

4.1 readme-team

A key part of what your teams submit is a **readme-*team*.pdf** that includes the following in this order:

1. The members of the team.
2. Approximately how much time was spent in total on the project, and how much by each student.
3. A description of how you managed the code development and testing. Use of github in particular is a particularly strong suggestion to simplifying this process, as is some sort of organized code review.
4. The language you used, and a list of libraries you invoked. For Arduinos, using the online IDE allows you to save your code to the cloud, and then if there is a common signin, all team members can work with it.
5. A description of the key data structures you used, especially for the internal representation of states and the state machines and the transitions.
6. Observations made during the project. For example, if an *nfa2dfa* was written, a discussion of what you observed in expansion of machine description complexity from NFA to DFA.
7. If you did any extra programs, or attempted any extra test cases, describe them separately.

Only one member of the team need submit this report to their Sakai directory.

4.2 teamwork-netid

In addition, each team member should prepare a brief discussion of their own personal view of the team dynamics, and stored in a PDF called **teamwork-*netid*.pdf**, where netid is your netid. The contents should include:

1. Who were the other team members.
2. Under whose netid is the **readme-*team*.pdf**, code, and other material saved.
3. How much time did you personally spend on the project, and what did you do?
4. Which machines did you design and how did you generate test strings?
5. What did you personally learn from the project, both about the topic, above programming and code development techniques, and about algorithms.
6. In your own words, how did the team dynamics work? What could be improved? (e.g. did you use github and if so did it help, did you meet frequently enough, etc.)
7. From your own perspective, what was the role of each team member, and did any member exceed expectations, or vice versa.

Each student's submission here should be in their own words and SHOULD NOT be a copy of any other team members submission, nor should they be shared with the other team members. These reports will be kept private by the graders and instructor, and will be used to ensure healthy team dynamics. The instructor retains the right to adjust the score of an individual team member from the base score (both up and down) on the basis of these reports. Also, a composite of all such reports from all projects will be used to create an overall "lessons learned" at the end of the project in what techniques seemed to work better, and where problems arose. These hopefully will be of use for the next project.

5 Submission

- Each team member should have in their own Sakai directory for this course a directory called Project2, where all submissions should go.

- When the team is ready to submit, one (and only one) team member will place copies of all code at output in the designated common directory.
  - Your non-Arduino code should be runnable on any of the studentnn.cse.nd.edu machines so that if there is an issue the graders can run the code themselves.
  - If you wrote in a compiled language like C++, include all needed source files (excepting standard libraries), a make file, and a compiled executable. The source code is there to allow the graders to look at the code for comments and to resolve any discrepancies that may arise in looking at your results.
  - If you wrote in a language like Python make sure your code is compatible with one of the versions supported on the studentnn.cse.nd.edu machines, again to allow the graders to check something if there is an issue.
  - If you developed code for an Arduino, include all sketches.
- Also in the same directory as the code the team should place an output file for each of the test files that you ran. The format should be as described in Section 3.1.2, and the name should be the same name as the test file but with a ".csv"

In addition, every team member should include in their own Project2 Sakai directory a copy of their **readme-*team*** file, again in pdf.

For the team members who did not upload code and readme's

- Machine and test files for the designs that the student did themselves.
- The output files from running the above machine files against the developed test files.
- A short readme describing what the machines are, what language they were supposed to accept, and whether or not the test set showed proper operation.