CSE 232 Spring 2017

Programming Project #9

Assignment Overview

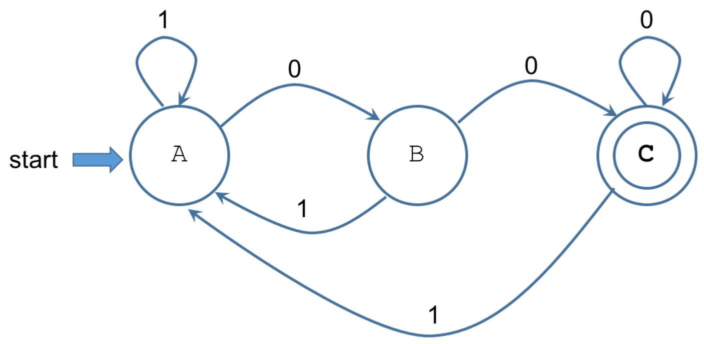
This project focuses on the use of classes. It is worth 50 points (5% of your overall grade). It is due Monday 4/10 before midnight. That's ***two weeks*** because of the midterm on 3/29.

The Problem

If you have a simple little algorithm that controls something like an elevator, or an ATM, or even a simple game AI, you can use a data structure called a Finite State Automaton/Finite State Machine or (FSA/FSM), see https://en.wikipedia.org/wiki/Finite-state_machine

The idea behind an FSA is simple. You have a series of states that are connected together by arcs forming a graph. Your algorithm begins at a designated state and reads an input symbol. The arcs connecting to other states are based on the existence of a particular input. That is, if you are in state A, you can move to state B only if the input has the symbol "X", or your can move to state C if the input has a symbol "Y". Similarly for other states and symbols. You can represent your simple algorithm as moving between states based on inputs.

Here's a simple example:



There is a designated start state (here A) and a designated success state, (here C). Given a set of inputs consisting of only 0's or 1's, what kind of string does this FSA represent. It is any binary string that ends in two or more 0's in a row. Let's try it. Here's an input string 101001000

start at A:

- on a 1, go to A
- on a 0, go to B
- on a 1, go to A
- on a 0, go to B
- on a 0, go to C
- on a 1, go to A
- on a 0, go to B
- on a 0, go to C
- on a 0, go to C

We start at A, consuming all the input we end up at C. If the FSA can transition from the start state to the end state having consumed all the input, then we say the FSA **accepts** the input successfully. The FSA is termed an **acceptor**, as it can accept/reject an input. For example, on the input 10101, the FSA would end up in state A after consuming all the input. State A is a the success state, so the FSA rejects that input as invalid.

An FSA can be generalized to take various inputs (coins into a vending machine, moves in a game), but for the sake of our little experiment we are going to stick with simple binary input, only 0 or 1.

Basic Premise
Though you can represent an FSA many ways, the core idea is to represent the transition table. The transition table represents the following information: if you are in a particular state and see a particular input, you would transition to this next state. For example, working with the diagram above, the transition table for state A would have the following information:
A: on input 1 go to A
: on input 0 go to B

We can represent the table fairly straight forwardly with the following type:
```
map<string, map<string, string> >
```

We interpret this type as follow:
- The key in this map is a state, the present state we are interested in
    - the map associated with this key contains a map where:
        - the key is an input
        - the value is the next state

Thus our table for ***just*** the state A would be
```
map<string, map<string, string>> trans_table={"A",
                                              { {"1", "A"}, {"0", "B"} }
                                              };
```
Can you read that? The whole trans_table is a map. The only key in trans_table is "A". The value associated with "A" is another map. This associated map has two keys, "1" and "0". Associated with each of those keys is the state you would go to next if you get that input. The lovely thing about this data structure is that finding the next state is pretty simple:

```
string next = trans_table["A"]["0"];
```

The key "A" yields a map. We index into that map with "0". Associated with "0" is the next state, "B";

Class FSA
The below is the provided header file for the class. As a general rule, the elements (states and inputs) are all strings, even though there may be cases where only a character is needed. If you need to convert a character to a string, remember the following constructor

`string(1, c)` creates a string of length one using the character c.

We will also be throwing `domain_error` under various conditions, for example:
- a state is not found in the fsa (and was supposed to be)
- a state already exists upon insertion into an fsa
- an input was not found as a valid transition

The exact string thrown depends on the situation, so look at the test cases for detail.

```
class FSA{
private:
  map<string, map<string, string>> table_;
  string state_;
  string start_;
  string finish_;

public:
  FSA()=default;
  FSA(string strt, string stp);
  FSA(ifstream&);
  bool exists_state(string);
  void add_state(string);
  void add_transition(string, string, string);
  string transitions_to_string(string);
  string next(string,string);
  bool run(string);
  friend ostream& operator<<(ostream&, FSA&);
  // string unreachable();  // something to try, not required
};
```

The private elements are:

- the designated start state `start_`
- the designated success state `finish_`
- `state_` represents which state the fsa is presently in as it traverses the fsa

The methods are:

- default constructor (done for you ☺)
- 2 arg constructor, which sets `start_`, `finish_` and `state_` (`state_` same as `start_`). Empty `table_`.
- constructor which takes an open `ifstream` (opened in main before call). Format is (see tests):
  - start state finish state
  - each subsequent line is an entry for table_ as: state1 input state2 (interpreted as, starting at a state1, given an input, transition is to state2)
- `bool exists_state(string s)`. Is the state s in the fsa (that is, is there a key in table_ that is equal to s).
  - note this is only for a key in the table_ itself, not in the map associated with state)
- `void add_state(string s)`. Adds s as a key to table_. The map associated with s is set to empty.
  - Throws domain_error if s is already in table_
- `void add_transition(string s1, string input, string s2)`. Updates table_. Interpretation is that existing state s1 is updated with transition to s2 on input
  - throws domain_error if s1 does not already exist in table_
- `string transitions_to_string (string s)`. Returns a string of all the transitions associated with the state s. See the test cases for format
  - throws domain_error if s is not already in table_
- `string next(string s, string input)`. Starting from state s and given input, return the next state.
  - to be general, the input is a string though logically with a sequence of 0,1 it might be a character. Convert (as noted above) to a string if necessary)

    - throws domain_error if state s doesn't exist in table_
    - throws domain_error if no transition on input exists from s in table_
- `bool run (string input)`. Starting from start state, run the fsa and, having consumed the input, return if the final state is the finish_ state or not
    - because you need to do it anyway, this method prints out a message to cout for each transition. Make sure you look at the test cases to get the format correct
    - obviously uses `next` to do the work.

The functions are:

`ostream& operator<<(ostream& out, FSA& fsa)` Prints a representation of the fsa including the start_, finish_ and state_ info and all the transitions (using `transitions_to_string`). See the test cases for examples. Returns ostream

Test Cases

Test cases are provided in the subdirectory `test` of the project directory (because it is getting confusing), at least one for each function . ***However!*** It is time you start writing your own tests. The tests I provide ***are not complete***, and we are free to write extra tests for grading purposes. Do a good job and test your code. Just because you pass the one (probably simple) test does not necessarily mean your code is fine. Think about your own testing!!!!

Assignment Notes

1. You are given the following files:
    a. `main.cpp` – This file includes the main function where the test cases will be run. **Do not modify** this file.
    b. `functions.h` – This file is the header file for your functions.cpp file. **Do not modify** this file.
    c. input#.txt – These are text files that will be used to run the test cases.
    d. correct_output_#.txt – These text files will be used to grade your output based on the corresponding input files. Be sure that your output matches these text files exactly to get credit for the test cases.
2. You will write only `functions.cpp` and compile that file using `functions.h` and `main.cpp` in the same directory (as you have done in the lab). You are only turning in `functions.cpp` for the project.
3. Comparing outputs: You can use redirected output to compare the output of your program to the correct output. Use redirected output and the "diff" command in the Unix terminal to accomplish this:
    a. Run your executable on the desired input file. Let's assume you're testing input1.txt, the first test case. Redirect the output using the following line when in your proj06 directory: `./a.out < input1.txt > output1.txt`
    b. Now your output is in the file output1.txt. Compare output1.txt to correct_output1.txt using the following line: `diff output1.txt correct_output1.txt`
    c. If the two files match, nothing will be output to the terminal. Otherwise, "diff "will print the two outputs.

Deliverables

`functions.cpp` -- your completion of the functions based on `main.cpp` and `functions.h` (both provided).

1. Remember to include your section, the date, project number and comments.
2. Please be sure to use the specified file name, i.e. “functions.cpp”
3. You will electronically submit a copy of the file using the "handin" program: http://www.cse.msu.edu/handin/webclient