CS 345 (Fall 18): Analysis of Discrete Structures

Project #6

Turing Machines

due at 5pm, Tue 4 Dec 2018

1 Introduction

In this project, you'll be implementing a Turing Machine simulator. It will be, in a lot of ways, pretty similar to the Dijkstra's Algorithm solution: my `TestDriver` will parse a `.dot` file, and call functions in your class to build the matching graph in memory; then I will tell you to run the simulator.

Of course, in a Turing Machine, each edge of the graph (known as a "transition" in TM lingo) has more parameters: instead of simply having a weight, each now has a read character, a write character, and a move direction.

Also, in the Dijkstra's Algorithm project, we had you save the state of the algorithm inside your graph, since we would perform the calculation in one step, and then print it out later. The reverse is true in this project; you **must not** save the state of the algorithm in the graph itself. This is necessary because you will be running the algorithm multiple times on the same graph. (We might even run it with different start state each time, in some testcases.)

2 What I Provide

As before, I've provided a test driver class, a common interface for the example and student code (`Proj06_TuringMachine`), and `.class` files for my example implementation.

You can see some example machines in the project directory; the simplest is the `hasOne` machine, and `equal_0s1s` is a little more complex. `bubbleSort` is also interesting - but is a lot more complex.

Each testcase has, associated with it, a `.inputs` file, which lists a set of example inputs to test (expect me to add more TMs, and more inputs to each TM, when I grade your code).

3 Implementation Details

3.1 Understanding the Output

Here's one step of the output from my example code:

```
aabcad
  ^   state: scanB_r
```

The first line tells you the current state of the tape; each character of output represents one character of the tape. (We only allow lowercase characters, digits,

and periods in our output, so it's always easy to read the output - you never have to worry about whitespace.)

Underneath that, is a caret which shows the current state of the head in the machine. In this case, the head is "on" the c - which means that, in the next step of execution, we will **read** the character c. Finally, the current state is printed: the string `scanB_r` is one of the states in the TM. (Take a look at `test_bubbleSort.dot` to see the code for this TM.)

3.2 Running Off The End Of Tape

If the TM runs off off the end of the tape (in either direction), you should **add a period character** to the end of the state. You should do this **any number of times.** Conceptually, the tape of any TM is an **infinite string** of characters - with the first few ones set to interesting information, but with an infinite stretch of "blank" characters, stretching out in both directions.

Of course, don't add additional characters to the string until the machine actually runs off the end - otherwise, you won't match the example output.

Also, don't ever worry about "cleaning up" old period characters. Although that might be a cool feature in a real TM, it isn't part of this project. Instead, your code should **extend but never shorten** the state string.

3.3 Don't Worry About Nondeterminism

You may design your Turing Machine to be deterministic; we will not test you on any non-deterministic machines.

But what, exactly does that mean? What can you assume is true about the various transitions in the graph? I'll let you all chew on that for a bit...

3.4 `debug`

The `run()` method of the common interface includes both a start string (the initial state of the tape before the TM starts) and a `debug` parameter. If `debug=false`, then you will print out a very little bit of information: you will simply print out the initial state, the ending state, and a message about whether the machine accepts or rejects (plus some newlines). Take a look at the example output - and compare it to the last few lines of the `main()` function in `TestDriver` - to see the exact format.

4 How to Run the Test Driver

4.1 Command-Line Arguments

The `TestDriver` accepts up to 2 arguments. The first is optional, and the second is required.

The first argument, if provided, is simply "example". If you provide this, then the `TestDriver` will run the example code; if not, it will run your code.

The second argument is the start string for the machine. It may include only lowercase letters, digits, and periods, and it must **not** be the empty string. The TM will always start with the head over the first character in this string.

5 Tips, Tricks, Hints, and Freedoms

5.1 Java Collection Classes

You may use Java collection and utility classes (such as `ArrayList`, `Set`, `HashMap`, etc.) as often as you want. I **encourage** you to use generics to make them easy to use - for instance, my solution used several `HashMap`s that mapped `String` to other types.

5.2 Classes

Your solution must include as many classes as you want; make sure to turn in all of the `.java` files to D2L.

6 Base Code

Download all of the files from the project directory

`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj06/`

If you want to access any of the files from Lectura, you can also find a mirror of the class website (on any department computer) at:

`/home/russelll/cs345f18_website/`

7 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

7.1 Testcases

You can find a set of testcases for this project at
`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj06/`

See the descriptions above for the precise testcase format, and also for information about how to run the two test driver classes.

7.2 Other Testcases

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcaes of your own, in order to better test your code.** You are also encouraged to share your testcases on Piazza!

7.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj06`. Place this script, all of the testcase files, and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

8 Turning in Your Solution

Turn in all of the .java files associated with your solution. Do not turn in any copies of the files that I have provided you.

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.