Computer Science 241

Program 2 (15 points)
Due Monday, May 16th, 2016 at 11:59 PM

Read all of the instructions. Late work will not be accepted.

Overview

For the second programming assignment you will work on your own to create a speech recognition program. Specifically, it is capable of so-called "second pass" decoding, which takes as its input a special directed acyclic graph called a *lattice*. Each node in a lattice corresponds to a point in time during a sentence, and each edge corresponds to a word that might have been uttered between the two end-points. Each path from the special start node to the special end node corresponds to one possible *hypothesis* for the sentence that was uttered. Each edge is weighted by two scores: an acoustic model score (amScore) and a language model score (lmScore). You can combined these into a single score by taking a weighted sum, weighted by a parameter called the language model scale (lmScale). The combined score for an edge e is:

$$combinedScore(e) = amScore(e) + lmScale * lmScore(e)$$

The score of a path is just the sum of the scores of its edges. These scores represent negative log probabilities, so the lower the score the better. The shortest path, then, is the best path (lowest score means highest probability). Conveniently, there are very efficient algorithms for computing the shortest path through a directed acyclic graph. For example, even in a graph with 9.518×10^{86} unique paths (i.e. possible hypotheses), the best hypothesis can be found in less than two seconds on a standard laptop.

You will implement this functionality in the `Lattice` class. A skeleton `.java` file with several dummy methods have been provided to you to complete. The pre- and post-conditions for each method are specified; it will be your job to complete all of the methods according to those conditions. Additionally, `Lattice` relies upon two classes that have already been implemented for you: `Hypothesis`, for representing and performing operations on a speech recognition hypothesis, and `Edge`, which is used to store the information associates with each edge in the lattice. You are welcome to add as many `private` helper methods as you would like.

Development and Testing

A program named Program2 (`Program2.java`) has been provided to you. **Do not modify Program2's code.** This program drives your `Lattice` class: it reads in test lattices, computes their best hypotheses, and reports various statistics of the lattices. It accomplishes this by generating new instances of type `Lattice`. Because `Program2` interacts with your `Lattice` class, you must not change the method header for any of the public methods in `Lattice.java` (including, for example, adding new `throws` flags). You must supply this

program with three arguments: the name of a "lattice list,"[1] an lmScale value for combining the acoustic and language model scores into a single combined score, and the name of an (**already created**) output directory, where two different representations of the lattice will be written. An example usage is:

```
C:> java.exe Program2 someLatticeListFile.txt 8.0 someOutputDirectory
```

or (in Linux):

```
$ java Program2 someLatticeListFile.txt 8.0 someOutputDirectory
```

I will use Program2 for grading; it is available for you to use during development. Close to the deadline, I will post one example output produced by my code so that you can check that your formatting is correct. You should develop at least two new, simple, lattices and use these as test cases.

Visualizing Lattices

For each `.lattice` file in the input lattice list, two files will be created in someOutputDirectory: 1) another `.lattice` file that should be in the same format as the input `.lattice` file,[2] 2) a `.dot` file that can be compiled into a pdf or svg image using the Graphviz `dot` program. At least for small lattices, the graphical representation can be a very helpful tool for understanding the data. To compile a `.dot` file into a pdf in Linux, use the following command

```
$ dot -Tpdf inputFileNameGoesHere.dot -o outputFileNameGoeshere.pdf
```

Simply replace `pdf` with `svg` everywhere it occurs to compile to svg format. You can view `svg` files with the `inkscape` program. Be warned that for very large lattices, the `dot` command can take a prohibitively long time to run.

Grading

Submitting your work

Submit a zip archive on Canvas. Your archive should have in it at least the following files:

- `Lattice.java`
- Your write-up (named `writeup.txt`)
- At least two new test input pairs you have created
  - Name your new test files `test1.lattice` (and `test1.ref`), `test2.lattice` (and `test2.ref`), etc.
  - Create a test list named `test.list` that contains all test lattices and refs
- Any other source code needed to compile your program / classes

[1] A lattice list simply contains a list of lattice files to process, along with their corresponding reference files. The file has one line per utterance. Each line begins with the name of a .lattice file, followed by a space, followed by the name of the corresponding .ref file.

[2] In practice, with the example files I provide, it should be *identical* to the input .lattice files.

Your archive need not and **should not contain your .class files.** Upon checking out your files, I will replace your version of `Program2.java`, `Edge.java` and `Hypothesis.java` with my original ones, compile all `.java` files, run `Program2` against a series of test lattices, analyze your code, and read your documentation.

Points

This assignment will be scored by taking the points earned and subtracting any deductions. You can earn up to 50 points.

Component	Points
Write Up & Test Cases	1
~~`Lattice constructor`~~	~~2~~
~~`Lattice getUtteranceID`~~	~~0.5~~
~~`Lattice getNumNodes`~~	~~0.5~~
~~`Lattice getNumEdges`~~	~~0.5~~
~~`Lattice toString`~~	~~0.5~~
`Lattice decode`	2
`Lattice topologicalSort`	2
`Lattice countAllPaths`	1
`Lattice getLatticeDensity`	1
~~`Lattice writeAsDot`~~	~~1~~
~~`Lattice saveAsFile`~~	~~1~~
`Lattice uniqueWordsAtTime`	1
`Lattice printSortedHits`	1
Total	**15**

You may also have deductions from your score for

- Poor code style (e.g. inconsistent indentation, non-standard naming conventions, excessively long methods, code duplication, etc.)
- Inadequate versioning
- Errors compiling or running

Write-Up & Test Cases

You need to create, add and commit a *plaintext*[3] document named `writeup.txt`. In it, you should include the following (numbered) sections.

1. Your name
2. Declare/discuss any aspects of your code that is not working. What are your intuitions about why things are not working? What potential causes have you already explored and ruled out? Given more time, what would you try next? Detailed answers here are critical to getting partial credit for malfunctioning programs, and failure to disclose obvious problems will lead to additional penalties.

[3] E.g. created with vim, kate or gedit.

3. An acknowledgment and discussion of any parts of the program that appear to be inefficient (in either time or space complexity).
4. A discussion of the portions of the assignment that were most challenging. What about those portions was challenging?
5. A discussion on how you approached testing that your program was correct and asymptotically efficient. What did `test1.lattice` test? What did `test2.lattice` test?
6. What was the most challenging aspect of this assignment, and why?
7. (Optional) Any other thoughts or comments you have on the assignment and its implementation.

Lattice Format

The provided lattices are in a very simple format. An example complete lattice file is listed below:

```
id CMU_20020319-1400_l01_denise_1_0200626_0201468
start 0
end 6
numNodes 7
numEdges 7
node 0 0.00
node 1 0.00
node 2 0.00
node 3 0.19
node 4 0.19
node 5 0.62
node 6 0.83
edge 0 1 -silence- 0 78
edge 0 2 -silence- 0 38
edge 1 3 -silence- 2325 0
edge 2 4 -silence- 2325 0
edge 3 5 ok 7414 12
edge 4 5 okay 7414 7
edge 5 6 -silence- 862 0
```

Each line begins with one of the following keywords:

- `id`. The next token after this keyword will be the identifier for the utterance (a string).
- `start`. The next token after this keyword is the index of the special start node (an integer). This is usually 0, but need not be.
- `end`. The next token after this keyword is the index of the special end node (an integer). This is usually the node with the largest index in the graph, but need not be.
- `numNodes`. The next token after this keyword is the number of nodes in the graph (an integer).

- `numEdges`. The next token after this keyword is the number of edges in the graph (an integer).
- `node`. This defines one node in the graph. The first token after the keyword gives the node index (an integer); the second token after the keyword gives the time associated with the node (a double).
- `edge`. This defines one edge in the graph. The first token after the keyword gives the source node index (an integer) and the second token gives the destination node index (an integer). For example, `0 5` indicates that this is an edge from node 0 to node 5. The third token after the keyword is the word/label associated with the edge (a string). The fourth and fifth tokens are the acoustic and language model scores (both integers), respectively.

Dot Format

There are many kinds of graphs that can be described in dot format. Ours will be simple. The files you produce will consist of three parts:

1. It will always begin with this header:

   ```
   digraph g {
       rankdir="LR"
   ```

2. It will be followed with edge definitions, one per edge in the graph. These take the form:

   ```
       srcNode -> destNode [label = "labelGoesHere"]
   ```

   For example:

   ```
       0 -> 1 [label = "-silence-"]
   ```

3. It will always end with a single right curly brace (to close the one opened in step 1):

   ```
   }
   ```

An example complete dot file for a simple lattice with only seven edges is given below:

```
digraph g {
    rankdir="LR"
    0 -> 1 [label = "-silence-"]
    0 -> 2 [label = "-silence-"]
    1 -> 3 [label = "-silence-"]
    2 -> 4 [label = "-silence-"]
    3 -> 5 [label = "ok"]
    4 -> 5 [label = "okay"]
    5 -> 6 [label = "-silence-"]
}
```

Academic Honesty

You must not share code with any of your classmates: you must not look at others' code or show your classmates your code. You cannot take, in part or in whole, any code from any outside source, including the internet, nor can you post your code to it. If you need help from a classmate, step away from the computer and *discuss* strategies and approaches, not code specifics. You must also not reveal the details of your "count all paths" algorithm to any of your classmates. I am available for help during office hours as are the department tutors. I am also available via email (do not wait until the last minute to email). If you participate in academic dishonesty, you will fail the course.