Computer Science 241

(Pair) Program 2 (50 points + 3 extra credit points)
Due Tuesday, Nov 19, 2013 at 10:00 PM
See Canvas for information about an optional submission on Tuesday, Nov 12

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

Overview

For the second programming assignment you will work with your assigned partner 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. 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 two new classes: `Lattice` for representing and performing operations on lattices, and `Hypothesis` for representing and performing operations on a hypothesis. Skeleton `.java` files with several public dummy methods have been provided for both. The pre- and post-conditions for each method are specified; it will be your job to complete all of the public methods according to those conditions. Additionally, these will rely upon a `Edge` class that has already been implemented for you. You are free to add as many `private` helper methods as you would like.

Pair Programming

As in Program 1, this is a pair programming assignment. You must work together with your assignment partner to design, develop and debug your work for Program 2. Recall that at any given moment when coding, one person (the "driver") will be writing the code, while the other will be reviewing and offering suggestions (the "navigator"). You and your partner should switch roles frequently. To succeed, you must both be on the same page and both be engaged in the development of the program. No designing, development or debugging

can be done without both partners present. The goal of this approach is to increase your productivity and expose you to new approaches to developing code. Often the same bug that would take you 3 bleary-eyed hours to hunt down alone can be found within minutes by a second pair of eyes. Be patient: there is plenty of time allotted to complete this assignment as long as you proceed at a steady pace. Ask for help from me or the department tutors if you need it.

I will assign your partner via Canvas. There is no lab, so you will need coordinate with your partner to find times when you can both be present. You will need to contact me ASAP if you cannot find any overlap in your schedules to collaborate. Two class days will be available for you to work with your partner: Monday, November 4th and Tuesday, November 5th. Although, alone, they will not be enough time to finish the assignment, please take advantage of them. **Let me stress again that no designing or programming is to be done without both partners present! If I determine this happened I will fail you and your partner for this assignment.**

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` and `Hypothesis` classes, reading test lattices, computing their best hypotheses, and reporting various statistics of the lattices. It accomplishes this by generating new instances of type `Lattice` and `Hypothesis`. This is in contrast to `Program1`, where your `LanguageLearner` class was used to store a collection of related static methods. Because `Program2` interacts with your classes, you must not change the method header for any of the public methods in either class (including adding new `throws` elements). 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. The results produced by my completed version on a specific set of test files will be available to you. Be sure to compare the output of your code against the these files to make sure that your formatting is identical (and that your output is correct). You should test your class on at least one new, very simple, lattice file that you create, so that you can carefully analyze the behavior of your program.

[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.

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 *identical* to the input `.lattice` file, 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, 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 very long time to run.

The Repository

You will use the same Git version control repository that that you used for Program 1. The only difference is that this time you will put your code in a subdirectory named `prog2` (capitalization, spacing and spelling matter!), rather than `prog1`. Of the two partners, pick one who will house the repository and do not create a `prog2` directory in the other's repository.[2]

You *must* use this repository for the development and submission of your work (failure to have **8 or more** non-trivial revisions will result in **non-trivial penalties**). If your repository was not used for Program 1 but will be used for Program 2, verify that you are able to access and interact with it *as soon as possible* and contact me if there is a problem.

Grading

Submitting your work

When the clock strikes 10 PM on the due date, a script will automatically check out the latest pushed version of your assignment. **(Do not forget to push the work you want submitted before the due date!)** Your repository should have in it, at the least:

- `Lattice.java`
- `Hypothesis.java`
- Your write-up
- At least one new test input file you have created
  - Name your new test files `test1.lattice` (and `test1.ref`), `test2.lattice` (and `test2.ref`), etc.
- Any other source code needed to compile your program / classes

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

[2] I will grade the one that has a prog2 directory.

Points

This assignment will be scored by taking the points earned and subtracting any deductions. You can earn up to 50 points, plus 3 extra credit points:

Component	Points
Write Up & Test Cases	5
`Lattice constructor`	7
`Lattice getUtteranceID`	1
`Lattice getNumNodes`	1
`Lattice getNumEdges`	1
`Lattice toString`	3
`Lattice decode`	7
`Lattice topologicalSort`	7
`Lattice countAllPaths`	3 (extra credit)
`Lattice getLatticeDensity`	2
`Lattice writeAsDot`	3
`Lattice saveAsFile`	1
`Hypothesis constructor`	1
`Hypothesis addWord`	3
`Hypothesis getPathScore`	1
`Hypothesis getHypothesisString`	2
`Hypothesis computeWER`	5
Total	**50 + 3 extra credit**

Several deductions may decrease your score, with penalties up to those listed below:

Deduction	Penalty
Poor code style	5
Inadequate versioning	**15**
Does not compile/run	**25**

Write-Up & Test Cases

With your partner, in one or two pages, provide a write-up of your implementation. Submit your writeup as a **plaintext** file[3] named **writeup.txt**. Please include all of the following:

1. The names of both partners.
2. The username for the student whose repository was used for Program 2.
3. **An acknowledgment and discussion of any parts of the program that are not working.**
4. **An acknowledgment and discussion of any parts of the program that appear to be inefficient (in either time or space complexity).**
5. A discussion of the portions of the assignment that were most challenging. What about those portions was challenging?

[3]E.g. created by Notepad, or vim, or emacs.

6. A discussion on how you approached testing that your program was correct and asymptotically efficient.
7. Your observations on the kinds of hypothesis sentences you see with a very low lmScale (≤ 3) vs a high lmScale (≥ 20).
8. Any other thoughts or comments you have on the assignment and its implementation.

Useful Algorithms

Topological Sort

A directed acyclic graph can be interpreted as describing a partial ordering of the nodes. For example, consider a graph of courses, with arrows from course A to course B indicating that A is a prerequisite for B. There will be courses that are sequentially constrained; e.g., CSCI 145 must come before CSCI 241. Other can be taken in parallel; e.g., CSCI 241 and CSCI 301 can be taken at the same time. You can say that 145 comes before 241, but we cannot say that 241 comes before 301 or than 301 comes before 241 (in terms of prerequisites). Topological sort produces an ordering of the nodes so that the partial ordering is satisfied. Every node in a topological sorting comes *before* the nodes in its adjacency set (put differently: every node comes *after* all of its prerequisites). If you think of an edge from A to B as indicating that B *depends* on A, then if you visit nodes in their topologically sorted order, then at any given time you will have already processed all of the nodes the current node depends on. That is a very useful property that will be needed later in the single-source shortest-paths algorithm for directed acyclic graphs. The topological sort algorithm is given in Figure 1.

Single-Source Shortest Path in a DAG

To decode, we need to find the shortest path from the lattice's start node to the lattice's end node. The naive approach would enumerate all possible paths, comparing them each and finding the one with the best probability (i.e. shortest path). Unfortunately, this is not at all practical for common lattice sizes. Instead, we can find the shortest path through a directed acyclic graph using the property of *optimal substructure*. To illustrate this idea, assume we build a graph where the nodes represent large towns and cities in Northern America. The edges are weighted by the distance between the towns. Assume the node "Bellingham" is connected to only two other nodes: Vancouver (BC) and Seattle. The shortest path from some start node, say, New Orleans, to Bellingham, will necessarily pass through either Vancouver or Seattle. Furthermore, the shortest path will Bellingham will trace the steps of the shortest path from New Orleans to either Vancouver or Seattle. This is the notion of optimal substructure: getting to Bellingham can be seen as optimally solving two subproblems (getting to Vancouver and getting to Seattle), and then figuring out which is shorter: a) getting to Vancouver and then driving from Vancouver to Bellingham, or B) getting to Seattle and then driving from Seattle to Bellingham. More generally, in the shortest paths problems, you try to find the shortest path to some node A that has k incoming edges from k different nodes. The shortest path from the start node to A is the one where the combined cost of taking the optimal route to one of these k nodes and then going from that node to A is cheapest.

Now, the town/cities graph example is not acyclic (e.g. you can get from Bellingham to Seattle and back again). The problem is easier for directed acyclic graphs. You can perform a topological sort of the nodes and then, one by one, find the shortest paths from the start node to each of the other nodes in the graph. If you solve these problems in topological order, by the time you go to determine the shortest path to some node A, you will have already found the shortest paths to all nodes to which A is adjacent. So finding the shortest path is easy: just compare the shortest paths for each of the nodes to which A is adjacent, adding the cost to get from that node to A, and take the minimum. If you repeat this process for all nodes in topologically sorted order, you will find the shortest paths from the start node to every other node in the graph, including the end node. The pseudo-code for this algorithm is listed in Figure 2. For a graph with `n` vertices, you will keep track of the shortest paths in two n-dimensional arrays: 1) an array of floating point numbers, `cost`, that keeps track of the cost of the best known path from the start node to every other node, and 2) an array of node indices, `parent`, which stores the predecessor for each node along its best path (e.g. Bellingham's entry would store "Seattle" in the map routing example above).

Word Error Rate (WER)

"Word error rate" (WER) is a measurement of how well a hypothesized utterance matches the actual (reference) utterance. To compute WER, you must first perform a *minimum edit distance* alignment between the two sequences. For example, if the reference sequence is "i think computer science is fun" and the hypothesized sequence is "computing signs is such fun" then the first two rows of Table 1 show a minimum edit distance alignment. Given an alignment, you can count the number of errors made. Errors come in three forms:

1. Deletions: a word that is present in the reference is not aligned to anything in the hypothesis.
2. Substitutions: a word that is present in the reference is aligned to a different word in the hypothesis.
3. Insertions: a word that is present in the hypothesis is not aligned to anything in the reference.

The other possible outcome is a match, where a word in the reference is aligned to the same word in the hypothesis. Word error rate is defined to be:

$$WER = \frac{\#D + \#S + \#I}{N} = \frac{MinimumEditDistance}{N} \tag{1}$$

where $\#D$ is the number of deletions, $\#S$ is the number of insertions, $\#I$ is the number of insertions, and N is the number of words in the reference sequence. For the example in Table 1, we get $WER = (2 + 2 + 1)/6 \approx 0.833$.

Minimum edit distance is computed by constructing a matrix d that has $n + 1$ rows and $m + 1$ columns, where n is the number of words in the hypothesis and m is the number of words in the reference. The pseudo-code to compute the minimum edit distance is given in Figure 3.

i	think	computer computing	science signs	is is	 such	fun fun
D	D	S	S	M	I	M

Table 1: One minimum edit distance alignment between reference "i think computer science is fun" and the hypothesis "computing signs is such fun". The first and second rows show the alignment between the reference and hypothesis. The third row labels the deletions (D), substitutions (S), matches (M) and insertions (I).

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. It 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-"]
}
```

Extra Credit

You and your partner can choose to implement the extra credit method, `countAllPaths`. As specified in the post-conditions, it should return the number of unique paths through the lattice; i.e., the number of possible hypotheses. Since the number of paths can easily exceed the largest value representable by an `int` or `long`, `countAllPaths` returns a variable of type `BigInteger`.[4] One "obvious" strategy would be to perform a recursive traversal of the lattice. This, however, will be prohibitively slow for large lattices. To earn credit for this method, your algorithm should be no slower than $O(|V|^2)$ (and could be even faster if something other than an adjacency matrix representation was used). As is hinted at in the `.java` comments, you can devise an algorithm that is vaguely similar to the shortest paths one used in decode.

Academic Honesty

To remind you: aside from your designated partner, you must not share code with 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 and your partner need help from another pair, all involved should step away from the computer and *discuss* strategies and approaches, not code specifics. I am available for help during office hours, as are department tutors, but you should attend these hours with your partner. I am also available via email (make sure you and your partner is included in the email and do not wait until the last minute to email). If you participate in academic dishonesty, you will fail the course.

[4] See http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html.

```
TopologicalSort
   inDegrees := array of in-degrees, indexed by node
   S := set of nodes with zero in-degree
   while !S.isEmpty() do
      n = removeElement(S)
      add n to (end of) result
      for b in adjSet(n)
         inDegrees[b]--
         if inDegrees[b] == 0
            S.add(b)
         fi
      od
   od
   if sum(inDegrees) > 0 then
      has cycle!
   else
      return result
   fi
end
```

Figure 1: Topological Sort algorithm.

```
ShortestPath
   // initialize the costs
   for i = 0 .. n-1 do
      cost[i] = positive infinity
   od
   cost[startNode] = 0

   // search - find the correct values for cost and parent
   foreach node n in topologically sorted order do
      for i in 0 .. n-1 do
         if  weightOfEdge(<i,n>) + cost[i] < cost[n]
            // found a cheaper way to get to node n
            cost[n] = weightOfEdge(<i,n>) + cost[i];
            parent[n] = i;
         fi
      od
   od

   // backtrack to recover the best path from parent
   node := endNode
   while node != startNode do
      add node to start of results
      node = parent[node];
   od
   return the path defined by the sequence of nodes named results
end
```

Figure 2: Single Source Shortest Paths Algorithm for a Directed Acyclic Graph

```
MinimumEditDistance(hypothesis,reference)
   for i in 0 .. n do
      d[i][0] = i
   od
   for j in 0 .. m do
      d[0][j] = j
   od
   for j in 1 .. m do
      for i in 1 .. n do
         if hypothesis[i-1] equals reference[j-1] then
            d[i][j] = d[i-1][j-1]
         else
            d[i][j] = 1 + min(d[i-1][j], d[i][j-1], d[i-1][j-1])
         fi
      od
   od
   return d[n][m]
end
```

Figure 3: Minimum edit distance algorithm. Here we assume that hypothesis is an array of the words in the hypothesis sequence, indexed from 0 to $n-1$. reference is an array of the words in the reference sequence, indexed from 0 to $m-1$.