# Computer Science 241

(Pair) Program 1 (50 points)
Checkpoint Due Friday, April 27th, 2018 at 10:00 PM
Program Due Monday, May 14th, 2018 at 10:00 PM

**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 combine 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:

$$combinedScore(e) = amScore(e) + lmScale \cdot lmScore(e)$$

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 (see later in this document). For example, even in a graph with $9.518 \times 10^{86}$ 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. Now that you have a sense of what you will be doing, let's take a look at *how* you will be working on the assignment.

## Pair Programming[1]

Pair programming is a software development technique where two programmers work together in front of one keyboard. One partner types code while the other is suggesting and/or reviewing every line of code as it is being typed. The person typing is called the driver.

[1]These guidelines are based on a previous version developed by Perry Fizzano.

The person reviewing and/or suggesting code is called the navigator. The two programmers should switch roles frequently (e.g. every 20 minutes). For this to be a successful technique the team needs to start with a good program design so they are on the same page when it is finally time to start typing on the computer. **No designing or programming is to be done without both partners present!** Pair programming has been shown to increase productivity in industry and may well increase yours, but there are additional reasons it is being used in this class. First, working in teams is something that WWU CSCI graduates now working in industry frequently report that they wish they had more experience with. Second, working in pairs is a good teaching tool. Inevitably, in each pairing the partners will have different styles and abilities (for instance, one person may be better at seeing the big picture while the other is better at finding detailed bugs or one person might like to code on paper first while the other likes to type it in and try it out). Because of that you will have to learn to adjust to another person's style and ideally you will meet each other half way when there are differences in approach. It's important that each person completely understands the program and so both parties need to be assertive. Be sure to explain your ideas carefully and ask questions when you are confused. Also it is crucial that you 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, the TA or the department tutors if you need it.

You will be assigned a partner by me and will need coordinate with your partner to find times when you can both be present. You will need to contact me ASAP if there is any reason you will not be able to collaborate. **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 Program1 (`Program1.java`) has been provided to you. **Do not modify Program1'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 `Program1` 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,"[2] an lmScale value for combining the acoustic and language model scores into a single combined score, and the name of an output directory (that you created **before** running your program), where two different representations of the lattice will be written. An example usage is:

```
C:> java.exe Program1 someLatticeListFile.txt 8.0 someOutputDirectory
```

or (in Linux):

```
$ java Program1 someLatticeListFile.txt 8.0 someOutputDirectory
```

[2] 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.

I will use Program1 for grading; it is available for you to use during development. Close to the deadline, I may 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 lattices and use them as test cases.

## The Repository

You will develop this program under either your or your partner's Git version control repository hosted on Github (the same one used for labs). I repeat: only one partner should host the files for this assignment. Since you will be working together at all times on the assignment, it shouldn't pose any problems. For this assignment, put your code in a new subdirectory named `prog1` (capitalization, spacing and spelling matter!), rather than `prog1`.

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**).

## Plan

Prior to the checkpoint, you will construct a plan for the program, documented in a plaintext[3] file named `plan.txt`. Using proper spelling[4] and grammar, the plan should include the following numbered sections:

1. A one paragraph summary of the program in your own words. What is being asked of you? What will you implement in this assignment?
2. Your thoughts (a few sentences) on what you anticipate being the most challenging aspect(s) of the assignment.
3. A proposed schedule for when you and your partner will meet to complete this assignment (e.g. Mondays, Tuesdays and Wednesdays from 5-7pm until it's done).
4. Your proposed deadlines for major milestones to complete this assignment.
5. A list of at least three resources you plan to draw from if you get stuck on something.

## Checkpoint

Shortly after the assignment is posted (deadline listed at the beginning of this document), you will have a checkpoint to make sure you are on track. To satisfy the checkpoint, you need to do the following:

- Clone your repository (if not already cloned)
- Create your `prog1` directory
- Create, add, commit and push the following files:
  - `Lattice.java` (can be the skeleton I provided, for now)
  - `plan.txt` (the plan in the previous section)
  - `writeup.txt` (can be empty for now)

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

[4] Hint: to spell check on the commandline use "aspell -c plan.txt"

## Grading

### Submitting your work

When the clock strikes 10 PM on the due date, a script will automatically pull the latest committed version of your assignment. **(Do not forget to add, commit and push the work you want submitted before the due date!)** Your repository should have in it, directly in the `prog1` subdirectory, at least the following files:

- `Lattice.java`
- Your write-up (named `writeup.txt`)
- Your plan (named `plan.txt`)
- At least one new test input pair and list you have created
  - Name your new test file(s) `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
  - All of these test files should be in `prog1` itself, not a sub-directory
- Any other source code needed to compile your program / classes

Your repository need not and **should not contain your .class files.** Upon pulling your files, I will replace your version of `Program1.java`, `Edge.java` and `Hypothesis.java` with my original ones, compile all `.java` files, run `Program1` 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. To earn full credit for a method it needs to be *correct*, *clear* and *asymptotically efficient* (see syllabus for explanation).

| 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` | 5 |
| `Lattice getLatticeDensity` | 2 |
| `Lattice writeAsDot` | 3 |
| `Lattice saveAsFile` | 1 |
| `Lattice uniqueWordsAtTime` | 3 |
| `Lattice printSortedHits` | 4 |
| **Total** | **50** |

Common sources of deductions include

- Poor code style (e.g. inconsistent indentation, non-standard naming conventions, excessively long methods, code duplication, etc.)
- Incorrect output
- Errors compiling or running your code
- Inadequate versioning
- Code that is not *testable* (e.g. because an earlier error blocks `Program1` from getting to that test.

## Write-Up & Test Cases

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

1. Your name and your assigned partner's name
2. A discussion of any aspects of your code that are 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. 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? Did you employ any other testing strategies?

## 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 to 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).

## 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.

### 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,[5] 2) a `.dot` file that can be compiled into a pdf or svg image using the Graphviz `dot`

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

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.

## 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, including looking at others' code or showing your classmates your code, aside from your assigned partner. 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.

```
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 .. |V|-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 .. |V|-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