Dash Dot Dot Dot Dash

CSC 250 – 30 points

**Assigned:** 9 July 2014 **Due:** 16 July 2014

**Introduction**

In 1844, the first Morse code message was sent from Washington to Baltimore. According to the protocol, dots were 1 tick long, dashes 3 ticks, spaces between letters 3 ticks and spaces between words 7 ticks. For the trained translator, this made the message unambiguous.

But, what if there were no pauses for letters or words? Would it still be possible to decipher the message?

**Overview**

*Note: To fully comprehend the description below, it is recommended that you read up on Bigrams. Wikipedia has a brief but helpful article.*

In this lab, you will be given several Morse code messages to decipher. The following is the software contract:

public class ExhaustiveDecoder {

public List<String> decode(String message) { … }

}

To accomplish the task of decoding something terse like this:

.......---.----.-.....------..-.-.

You will need to perform an exhaustive search across the astronomical number of combinations which are potential decodings of this string. A common approach to exhaustive search is backtracking, which you can read about in your Algorithms book.

***Branching.*** You will have some libraries to help you along the way. Here is how they work:

DecodingDictionary dd = **new** DecodingDictionary();

ExhaustiveDecoder decoder = **new** ExhaustiveDecoder(dd, 100);

String[] possibilities = decoder.decode(".......---.----.-.....------..-.-.");

**for** ( **int** i = 0; i < possibilities.length; i++ ) {

System.*out*.println(i + ": " + possibilities[i]);

}

The DecodingDictionary is a dictionary that maps dictionary words to a specified encoding. By default, it encodes to Morse code. You will see that the Morse code class implements StringEncoder and that a great many more classes implement the same for different kinds of encoding. Using the dictionary to find words that are possible in the string can reduce branching a great deal.

Set<String> codes = dd.getCodes(); // gets the encoded dictionary words

Set<String> words = dd.getWordsForCode(code); // gets the words that share that encoding

The dictionary also provides some insight into the likelihood that one word will appear after another, which can further reduce branching.

int frequency = dd.frequencyOfFollowingWord(previousWord, word); // gets the frequency with with word follows previousWord in English sentences

DecodingDictionary uses a few helper classes that you may find profit in overriding for testing purposes. For example, here is another way to construct a DecodingDictionary:

DecodingDictionary dd = new DecodingDictionary(new MorseCodeEncoder(), new BigramDictionaryReader(new FileInputStream(“myverysmalldictionary.txt”)));

Making your own encoder (say, one that leaves the values the same) and your own dictionary (say, one that returns only a handful of bigrams) can make your testing go much faster and help you detect errors with much less pain.

***Scoring.*** Your code will exhaustively branch out to find various decodings of the Morse code messages provided, effectively abandoning branches early on that have no hope. There are two aspects to consider:

* *Bigram frequency.* Bigram frequency ranges from the low teens to the tens of thousands. Roughly speaking, this means that there are some word pairs that are thousands of times likelier than others. When confronted with very low likelihoods, it will be up to you to consider how to take advantage of it
* *Overall Likelihood.* Again speaking in approximations, the goodness of a sentence as a whole can be verified by taking all the set of frequency scores (n – 1 scores for a sentence of length n), normalizing them by dividing by a large number (like 10000), and multiplying them together, e.g.:

The role of the division is to model the probability of a chain of bigrams as a chain of independent statistical events. The 10000 acts as an approximation of the number of possible bigrams. *(Note: For the super-mathy folks in the class, the number of possibilities is actually much higher, of course; however, using the actual number takes the probabilities very close to zero, introducing potential loss of precision issues.)*

**Requirements**

* Your algorithm should use backtracking to decode the Morse code message. This means that the classification for your algorithm will be nearly factorial (ouch!).
* You must use only core Java other than the nulibraries.jar file and the commons-codec.jar given as part of the download.
* The ExhaustiveDecoder must working independently from your Driver (e.g., if I used my own Driver, I could point it at your decoder and things would still work)
* The dictionary is large and takes about 10 seconds to load. The messages given to you will not exceed 80 Morse Code characters. Aside from the dictionary load time (which should happen once at startup), your decode method should take no more than 30 seconds for an 80 character Morse Code.

**Now for some fun**

Here is a secret message for you to decode:

**....-.-.-.---..--.-..--..-...--..-.........**

**-.....-.-.----.........-----.-.-.-...-.--.**

**.--..-...-..-..----..**

**.--.-...--..-.-.--.-.----.**

Send each line in one at a time to your decoder. As part of your submission, submit what the secret message is.

Here are also some random movie quotes that you can test your decoder against:

**.......---.----.-.....------..-.-.**

* From Jerry McGuire

**...-...-..-....-....--.-.-.-.**

* From The Terminator

**.-......-........-....-.-...--...----..----..-.-.-.....----**

**.-.---.-...--....**

* From Forrest Gump

**-----....---.-.-.---------....**

- From Hamlet

**-.-.---..--.-..--.-.....--.-...-...-.....-.-...--....**

- From A Few Good Men

**...----.-.---..-.-...-..--......-.**

* From Star Wars

**What to Study**

It’s recommend that you study backtracking. Read up on the N-Queens problem. There are several examples online. Also Google for “bigrams” to learn about what they are and why they are useful.

**What to Hand In**

You will submit to your GitHub repo the following:

* At least one Java file, **ExhaustiveDecoder.java**. Please use the default package. Submissions whose files are named incorrectly or methods are named incorrectly will be dinged **one point** off and handed back for repair.
* A text entry which addresses the following points:
  + Send me what my message was to you
  + Give a detailed explanation of how your algorithm works.
  + Outline any help you received from classmates or other students
  + Mathematically analyze the performance of your algorithm using big-Oh notation. Show that your algorithm performs at or better than O(*n!*).
  + Empirically analyze the performance of your algorithm. Outline your approach (what made you decide these experiments were good? Etc.) and publish a table of your results, e.g.

Message Length Expansion Ratio Completion Time

--------------------- --------------------- -----------------------

45 250% 40s

35 250% 15s

9 300% 2s

Etc.

**How You Are Graded**

You will perform a pass/fail verbal defense of your code to me. You may do this at any time and as many times as you like up until the due date. This is worth **25%** of the lab grade.

For the remaining 75 percentage points: **30%** for correctly using backtracking, **10%** for valuable branching optimizations, **20%** for the detailed analysis of your algorithm, **10%** for coding style (overall neatness and professional look, etc.), and **5%** for informing me what my secret message was.

**Stretch Goals**

Try other encoders to see if your algorithm still works. While the ranking might be different, a correctly coded algorithm should still work with different encodings (consider DoubleMetaphone, which is what this lab was originally designed for).

Further pruning can be done by introducing parsing rules, e.g. don’t follow an article (a, the) with another article. Calculate your average branch factor at various points in the lab and demonstrate how you lowered it substantially. (+2)

Create a Random Code encoder that picks codes of random lengths on construction, assigning them to the letters of the alphabet. Characterize how performance changes based on how the codes are picked. (+2)

**Appendix**

*Decoding Using Backtracking (a brief overview)*

Consider the following message:

.--......-.......-.....-....--.....-.--------

Your code might begin by checking the codes in the dictionary to find possible candidates. It notices that the first character could be interpreted as the letter A, which makes a word.

.--......-.......-.....-....--.....-.--------

A

Okay, so far so good. Let’s move on:

Checking against the rest of the string, it notices that the next few characters match up with the encoding in the dictionary for the word BEER

.--......-.......-.....-....--.....-.--------

A B E E R

The algorithm asks the dictionary what the likelihood is that the word BEER follows A in the English language. The test passes the algorithms threshold, so it keeps it and moves on.

The algorithm continues until it gets to a spot in the Morse code where there are no more words that have a high enough likelihood to try out.

For brevity, let’s suppose that the algorithm maps out like this (it doesn’t, the message actually says something else):

.--......-.......-.....-....--.....-.--------

A B E E R I S A S I T S A T I E I N M O O

The phrase “A beer is as it’s a tie in moo” doesn’t seem terribly likely. What to do?

We can take the likelihoods that the dictionary gives us and multiply them together like so:

(made-up numbers follow)

1000 A Beer

2000 Beer Is

1400 Is As

800 As Its

4000 Its A

200 A Tie

450 Tie In

105 In Moo

So,

score = (1000/10000) \* (2000/10000) \* (1400/10000) \* (800/10000) \* (4000/10000) \* (200/10000) \* (450/10000) = 8.064E-08

The algorithm repeats this process with every combination that it finds and then picks the top 20, ranks them, and displays them on the screen.