Programming Assignment 3

EE360C

Report:

A. ALGORITHM DESCRIPTION

B. CORRECTNESS PROOF

C. LOOP INVARIANT

D. EXTRA NOTES

E. COMPILATION INSTRUCTIONS

George Neal

gln276

**A. ALGORITHM DESCRIPTION**

**Approach:**

My solution to lab 3 used a Dynamic Programming approach. Essentially, my algorithm breaks down the transcript, one valid entry at a time, until the transcript is empty. This is done recursively, so each time a solution is found, the algorithm returns true. If the function call sees that the result it true, then it maps the current transcript to the successful entry used as a prefix. If no entries led to an empty transcript, then false is returned. This mapping is accomplished very quickly. Next, I use an iterative algorithm to reconstruct all of the solutions from the memoization table that was created by mapping transcripts to valid entries that led to solutions. This reconstruction uses less iterations than the recursive function uses function calls, but still takes significantly longer in cases with huge inputs due to overhead. The pseudo code for this is shown below.

**Pseudo Code:**

decoderTableBuilder(String transcript)

{

// base cases

If (transcript is empty)

Return true

If (there exist solutions for this transcript already)

Return true;

// decomposition

For (each entry in the dictionary that is a prefix of the transcript)

String decomposedTranscript = transcript – prefix

If (decoderTableBuilder(decomposedTranscript) == true)

Map transcript to entry

Return true if **some** prefix led to a solution

Return false if no prefixes led to a solution

}

decoderSolutionBuilder(transcript)

{

Transcripts is a list of transcript strings

Solutions is a 2D array of solution strings for each transcript

TempTranscripts, and TempSolutions are used for manipulation

Initialization:

Transcripts = {transcript}

Solutions = {{empty string}}

While (transcripts is not empty)

{

For (each transcript of transcripts)

{

Prefixes = all prefixes this transcript maps to in table

For (each prefix in Prefixes)

If (prefix decomposes transcript to empty string)

Add all solutions for this transcript to solution list

Else

{

Add a list to TempTranscripts and TempSolutions

Add decomposed transcript to end of TempTranscript

For (each string in current Transcript’s solution list)

Add prefix to solutions and add to Temp Solutions

}

Transcript <- TempTranscript

Solutions <- TempSolutions

Clear TempTranscript and TempSolutions

}

}

**Time Complexity:**

o(n^n), closer to O(n \* (1 + average number of times that transcript was successfully broken down using more than one entry at any iteration of the recursive call), where n is the length of the string.

I say that the algorithm is O(n \* above number) because if the transcript was always broken down by n entries, then we could say that the time complexity is O(n^n); at each composition, we break the length n string into n subproblems, each of which will be further broken into n subproblems, etc. It will be broken down into n subproblems n times because in the worst case, it is broken down by one character each time. However, the number of subproblems it is broken into are much, much smaller; it validly occurs most often when the largest valid prefix can be broken into more than one other word, usually leading to at least 3 more subproblems.

The recreation of the memoized solution will have less iterations (rather than function calls) because it picks the valid ones every time (like a NP), but it seems to take significantly longer, because it must append prefixes to lists of long strings with sometimes MANY solution strings. In other words, it still has the same time complexity as above, if not less because we only count valid prefixes, but there is a time constant associated with the overhead at each iteration that makes it take much longer on large inputs (the more solutions, the longer it takes).

**B. PROOF OF CORRECTNESS**

We must prove that the algorithm above provides all possible such sentences that are composed of entries, which, without whitespace, equals the original transcript.

We can prove this by examining the pseudo code above.

First, we look at the base cases of the algorithm. These return the recursive call when the transcript to decompose is empty, i.e. it has already been fully decomposed, and when the transcript we are decomposing has already been decomposed previously. We know this because for each decomposed transcript we examine, we further decompose it using every valid dictionary as a prefix. If that prefix leads to a solution, we save this fact using the table, and can later use it to save time using the second base case. If no solution is reached for a transcript, the transcript is mapped to null in the source code. Because this algorithm decomposes the transcript using every possible valid entry as a prefix, we know that every combination of entries that lead to the completely decomposed transcript are saved in the table.

Now that we know all entries that lead to valid decompositions of the transcript are saved in the table, we just need to show that we properly build the solutions from them. We can see this by examining the pseudo code. We begin with a whole transcript and a set of empty solution string lists, one string list for every transcript. For each transcript we examine, we decompose it using one of the entries that was used as a prefix in the memoization table, save the decomposed transcript to a list of transcripts to examine on the next iteration, and append the prefix to all of the solutions of the transcript before it was decomposed by that prefix. This guarantees that we use every prefix used to decompose the problem in the recursive algorithm described above. Once the decomposed transcript is the empty string, we save all of the appended solutions to the list of solutions.

**C. LOOP INVARIANT**

The loop invariant of algorithm described in part A. is the relation where the solution of the currently examined transcript (a decomposition of the current transcript) concatenated with the decomposition of the entries that led to the current decomposition (sentence composed of dictionary entries separated by spaces) is equal to some solution of the transcript. We could also say that the loop invariant is the relation where the currently decomposed transcript concatenated to the end of the dictionary entries whose use as prefixes led to the decomposed transcript, and which are not separated by spaces, is the original transcript to be decoded.

**D. EXTRA NOTES**

1. When running the “a” input file with its transcript string appended to itself:
   1. I was able to get all 864^2 output values
   2. This took about 15 seconds on my computer (not including print time, which took a several minutes using the eclipse console output)
   3. I allocated A LOT of VM to this application. About 6GB using -Xmx6144M
      1. It is hard to say whether it used this much space: I looked at windows task manager and did not see any spikes above 3.7GB.

**E. COMPILATION INSTRUCTIONS**

Included is the RUN.jar executable java file, and my source code in a

 directory whose contents are formatted to that of an eclipse project.

Further Down are parts 1 and 2 (pseudo code and time complexity, and proof of correctness).

From here, the Java source code files can be reached in the directory

EE360Cprog3/src/assignment3

Example executable call

java -jar RUN.jar input.in

The executable will generate output in the Console.

Compilation:

I include the eclipse formatted directory because that is how I know to compile the code and create the executable. If the source is imported into the eclipse IDE, one can compile the code and export an executable as follows:

1. Right click the project in the package explorer pane

   Click "Export" from the context menu

2. Select Java/Runnable JAR file

   Click Next>

3. From the Launch Configuration drop down, Select Prog3Driver - EE360Cprog3

   Designate the Export destination, and include the executable name at

   the end of the directory path.

   Under Library handling, select Extract required libraries into generated JAR

   Do not save as ANT script

   Click Finish

NOTE: Prog3Driver.java contains the main method

The executable should now be in the designated directory from step 3.