Morse Code Lab Write Up

**Secret message**

"If you can read [this/these]"

"The code is working"

"Well done"

"And Party on"

**Algorithm Explanation**

*Input: A string message encoded in Morse code, without any spaces.*

*Output: A list of the top 20 sentences that could potentially be translated from the message.*

1. Initialize a series of variables to be passed to and used in the recursive method.
   1. A string representing the previous word in the sentence to be built is empty.
   2. A string representing the decoded sentence as it has been built so far is empty.
   3. A string representing the remainder of the Morse code message that has yet to be decoded is set to the Input message.
   4. A list of strings that will hold each built sentence is instantiated as empty.
2. Call the recursive method, which will perform the backtracking, passing it the variables containing the previous word, the decoded sentence that has been built so far, the remainder of Morse code that has yet to be decoded, and the list of strings that it holding each completed sentence.
   1. If the string containing the remaining Morse code to be decoded is empty, the end of this branch has been successfully reached. Add the decoded sentence to the list of complete sentences and return, else move to step 2.2.
   2. A positive integer i is set to 0.
      1. A substring of the remaining Morse code message is taken, the inclusive beginning index being 0, and the exclusive ending index being i+1.
      2. A call is made to the Decoding Dictionary, passing it the substring of Morse code and receiving a set of unique words that are valid translations for that piece of Morse code.
      3. If the set of words is not empty, that is, if the piece of Morse code successfully matched at least one word in the dictionary, move to step 2.2.3.1., else move to step 2.2.4.
         1. For each word in the set or matching words:
            1. If this word is the first word of the currently building sentence, that is, if the string containing the current sentence in progress is empty, OR if the frequency of this word when following the previous word in the sentence is greater than the required sentence, move to step 2.2.3.1.2., else iterate to the next word in the set.
            2. Create a copy of the current sentence as it has been built so far. Add this word to the end of that copy, followed by a space.
            3. Initialize a string to represent the new previous word, and set it equal to this word.
            4. Create a copy of the remaining Morse code that has yet to be decoded. Remove from this copy the piece of Morse code that was used to translate to this word.
            5. Move to step 2, using the newly copied variables and the current list of complete strings.
      4. Iterate i by 1. If i is less than the number of characters of remaining Morse code, return to step 2.2.1. Else, move to step 2.3.
   3. Return the list of completed sentences.
3. For each returned, completed sentence:
   * 1. A float representing this sentence's overall likelihood score is set to 1.
     2. For each word in the sentence, starting with the word at index 1 (the second word in the sentence):
        1. Take the frequency score of the current word following the previous word in the sentence. Divide this number by 1000. Set the likelihood variable by itself, times this calculated frequency score.
4. Sort the list of sentences by descending likelihood score.
5. Take the first 20 sentences, being the 20 most likely sentences in the list, and return them.

**Mathematical Analysis**

The majority of my algorithm's analysis will center around the actual recursive aspect of it, as that is what makes the biggest impact on the overall efficiency of the algorithm. Specifically, the number of times the recursion occurs, or the number of times the backtracking method calls itself, is what will actually determine the efficiency class.

n the perfect case scenario, the first time you called the recursive function it would run through every possible substring of the original Morse code message and use each of those substrings as the starting branch, the first word each sentence now being considered. This would assume that every substring actually translated to at least one word in the dictionary. So, your first branches would be using Morse code messages of character count n-1, n-2, … 2, and 1 characters long. If the algorithm proceeded this way, and each successive branch created branches of n-1, n-2, etc., then you would end up with an efficiency of around n!, as most exhaustive backtracking algorithms are known to be. However, this algorithm is not "perfectly" exhaustive, in that it has operations in place that can stop certain branches from being considered, stopping them short and not allowing the algorithm to continuing that branch's child branches and eventual leaves. The only reason each substring was able to proceed and start its own branch for consideration was because they were a part of the first level or branching - they make up the first words of the sentences to be built, so their frequency score cannot be checked against some previous word. Every successive branch, however, will have its frequency checked to be above a specific threshold, and, if it doesn't meet that requirement, that word will not be considered and that potential branching off would be stopped short. This doesn't even take into account the initial assumption I mentioned. Realistically there will be times that a substring doesn't even return translated words to be considered at all, cutting down the actual number of recursions, and therefore operations, even further.

Say the number of substrings of the current Morse code, being of length n, was n substrings. The number of words that would be considered, and therefore the number of next branches, would be cn, where c is the number of translations that substring receives. For the sake of this analysis, I will simplify this to c being 1, as the constant there will not make a major impact on the overall analysis. So essentially, I'm assuming a perfect case of n substrings, and each substring has exactly one word that it was able to translate to. Considering the things I mentioned in the paragraph above, the number of actual branches that would come of this would not be n substrings, but some variable k, where n minus k is the number of substrings that were ineligible to be considered. This means that the first level of branching would have a total of k branches being considered. Each of these branches would then branch off into k more branches each, so we have k\*k total branches. This would continue, each level of branching multiplying the total number of branches by k. However, the number of levels this could actually continue is limited by n. For instance, if n was 8, the highest number of substrings you could have (and therefore maximum total number of branches) is 8.

What this basically means is that the number of branches would be a factor of k, starting at k^0=1, the first time the recursive method is called by some other method other than itself, and then that exponent would iterate to k^1, k^2, etc. until k^n is reached, at which point no further branches could be made. Therefore, k^n is the maximum number of branches that could possibly result from the algorithm, k being the first number of words that were eligible to be considered, having come from k number of substrings, and n being the total number of substrings that could have been considered, rooting from the number of characters (n) in the Morse code message. This also means that k^n is the maximum number of times the recursion will run, and can be used to define an efficiency class for the algorithm.

In summary - the efficiency of this algorithm is T(n) = k^n, where k is the number of branches created when the recursive method is first run. This means that the T(n) making the algorithm's efficiency exponential which is, while still expensive, more efficient than a factoral efficiency.

Empirical Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Message Length | Threshold | Number of potential sentences created | Completion time (milliseconds) |
| 34 | 100 | 210 | 69 |
| 34 | 150 | 59 | 27 |
| 34 | 200 | 53 | 24 |
| 29 | 100 | 34 | 8 |
| 29 | 150 | 22 | 8 |
| 29 | 200 | 9 | 6 |
| 76 | 100 | 75 | 100 |
| 76 | 150 | 13 | 19 |
| 76 | 200 | 0 | 15 |
| 30 | 100 | 6 | 3 |
| 30 | 150 | 2 | 3 |
| 30 | 200 | 1 | 3 |
| 53 | 100 | 280 | 80 |
| 53 | 150 | 57 | 41 |
| 53 | 200 | 31 | 26 |
| 34 | 100 | 26 | 2 |
| 34 | 150 | 23 | 3 |
| 34 | 200 | 23 | 3 |

For this analysis I ran three separate tests on each of the 6 different quotes we were given on the worksheet. I chose to use those quotes because I had thoroughly tested with them previously, and knew that I would get the correct sentence back from each, that way I could save time by not printing every group of sentences out and picking through to make sure the proper one made it through. On each test I changed the threshold value that was used to filter the word combinations, that way I could not only see the difference the character count made, but the difference the threshold made as well.