CS5010 Programing Design Principles

Lecture 8 – An Example That Uses Several Maps

Tamara Bonaci

(Credit: An example from M. A. Weiss, Data Structure and Algorithm Analysis in Java, ch. 4, p 154-160.)

Many words are similar to other words. For instance, by changing the first letter, the word **wine** can become **dine, fine, line, mine, nine, pine,** or **vine**. By changing the third letter, **wine** can become **wide, wife, wipe,** or **wire**, among others. By changing the fourth letter, **wine** can become **wind, wing, wink,** or **wins**, among others. This gives 15 different words that can be obtained by changing only one letter in **wine**. In fact, there are over 20 different words, some more obscure.

We would like to write a program to find all words that can be changed into at least 15 other words by a single one-character substitution. We assume that we have a dictionary consisting of approximately 89,000 different words of varying lengths. Most words are between 6 and 11 characters. The distribution includes 8,205 six-letter words, 11,989 seven-letter words, 13,672 eight-letter words, 13,014 nine-letter words, 11,297 ten-letter words, and 8,617 eleven-letter words. (In reality, the most changeable words are three-, four- and five-letter words, but the longer words are the time-consuming ones to check.)

The most straightforward strategy is to use a `Map` in which the keys are words, and the values are lists containing the words that can be changed from the key with a one-character substitution. The routine in Figure 1 shows how the `Map` that is eventually produced (we have yet to write code for that part) can be used to print the required answers. The code obtains the entry set and uses the "enhanced" `for` loop to step through the entry set and view entries that are pairs consisting of a word and a list of words.

```
1       public static void printHighChangeables( Map<String,List<String>> adjWords,
2                                                int minWords )
3       {
4           for( Map.Entry<String,List<String>> entry : adjWords.entrySet( ) )
5           {
6               List<String> words = entry.getValue( );
7
8               if( words.size( ) >= minWords )
9               {
10                  System.out.print( entry.getKey( ) + " (" );
11                  System.out.print( words.size( ) + "):" );
12                  for( String w : words )
13                      System.out.print( " " + w );
14                  System.out.println( );
15              }
16          }
17      }
```

Figure 1 Code used to print out the related words

```
1       // Returns true if word1 and word2 are the same length
2       // and differ in only one character.
3       private static boolean oneCharOff( String word1, String word2 )
4       {
5           if( word1.length( ) != word2.length( ) )
6               return false;
7
8           int diffs = 0;
9
10          for( int i = 0; i < word1.length( ); i++ )
11              if( word1.charAt( i ) != word2.charAt( i ) )
12                  if( ++diffs > 1 )
13                      return false;
14
15          return diffs == 1;
16      }
```

Figure 2 Method to check if two words differ in only one character

The routine in Figure 2 is a straightforward function to test if two words are identical except for a one-character substitution. The issue, however, is how to construct the `Map` from an array that contains the 89,000 words. We can use the routine to provide the simplest algorithm for the `Map` construction, which is a brute-force test of all pairs of words. This algorithm is shown in Figure 3.

```
1       // Computes a map in which the keys are words and values are Lists of words
2       // that differ in only one character from the corresponding key.
3       // Uses a quadratic algorithm (with appropriate Map).
4       public static Map<String,List<String>>
5       computeAdjacentWords( List<String> theWords )
6       {
7           Map<String,List<String>> adjWords = new TreeMap<>( );
8
9           String [ ] words = new String[ theWords.size( ) ];
10
11          theWords.toArray( words );
12          for( int i = 0; i < words.length; i++ )
13              for( int j = i + 1; j < words.length; j++ )
14                  if( oneCharOff( words[ i ], words[ j ] ) )
15                  {
16                      update( adjWords, words[ i ], words[ j ] );
17                      update( adjWords, words[ j ], words[ i ] );
18                  }
19
20          return adjWords;
21      }
22
23      private static <KeyType> void update( Map<KeyType,List<String>> m,
24                                            KeyType key, String value )
25      {
26          List<String> lst = m.get( key );
27          if( lst == null )
28          {
29              lst = new ArrayList<>( );
30              m.put( key, lst );
31          }
32
33          lst.add( value );
34      }
```

Figure 3 A program to compute a map with words as keys, and values as list of words that differ in only one letter (the first approach).

To step through the collection of words, we could use an iterator, but because we are stepping through it with a nested loop (i.e., several times), we dump the collection into an array using `toArray` (lines 9 and 11). Among other things, this avoids repeated calls to cast from `Object` to `String`, which occur behind the scenes if generics are used. Instead, we are simply indexing a `String[]`.

If we find a pair of words that differ in only one character, we can update the `Map` at lines 16 and 17. In the private `update` method, at line 26 we see if there is already a list of words associated with the key. If we have previously seen `key`, because `lst` is not `null`, then it is in the `Map`, and we need only add the new word to the `List` in the `Map`, and we do this by calling `add` at line 33. If we have never seen `key` before, then lines 29 and 30 place it in the `Map`, with a `List` of size 0, so the `add` updates the `List` to be size 1. All in all, this is a standard idiom for maintaining a `Map`, in which the value is a collection.

The problem with this algorithm is that it is slow. An obvious improvement is to avoid comparing words of different lengths. We can do this by grouping words by their length, and then running the previous algorithm on each of the separate groups.

To do this, we can use a second map! Here the key is an integer representing a word length, and the value is a collection of all the words of that length. We can use a `List` to store each collection, and the same idiom applies. The code is shown in Figure 4. Line 9 shows the declaration for the second `Map`, lines 12 and 13 populate the `Map`, and then an extra loop is used to iterate over each group of words. Compared to the first algorithm, the second algorithm is only marginally more difficult to code and runs about five times as fast.

```
1       // Computes a map in which the keys are words and values are Lists of words
2       // that differ in only one character from the corresponding key.
3       // Uses a quadratic algorithm (with appropriate Map), but speeds things by
4       // maintaining an additional map that groups words by their length.
5       public static Map<String,List<String>>
6       computeAdjacentWords( List<String> theWords )
7       {
8           Map<String,List<String>> adjWords = new TreeMap<>( );
9           Map<Integer,List<String>> wordsByLength = new TreeMap<>( );
10
11            // Group the words by their length
12          for( String w : theWords )
13              update( wordsByLength, w.length( ), w );
14
15            // Work on each group separately
16          for( List<String> groupsWords : wordsByLength.values( ) )
17          {
18              String [ ] words = new String[ groupsWords.size( ) ];
19
20              groupsWords.toArray( words );
21              for( int i = 0; i < words.length; i++ )
22                  for( int j = i + 1; j < words.length; j++ )
23                      if( oneCharOff( words[ i ], words[ j ] ) )
24                      {
25                          update( adjWords, words[ i ], words[ j ] );
26                          update( adjWords, words[ j ], words[ i ] );
27                      }
28          }
29
30          return adjWords;
31      }
```

Figure 4 Method to compute a map containing words as keys, and lists of words that differ in only one letter as values (the second approach).

Our third algorithm is more complex, and uses additional maps! As before, we group the words by word length, and then work on each group separately. To see how this algorithm works, suppose we are working on words of length 4. Then first we want to find word pairs such as **wine** and **nine** that are identical except for the first letter. One way to do this, for each word of length 4, is to remove the first character, leaving a three-character word representative. Form a `Map` in which the key is the representative, and the value is a `List` of all words that have that representative. For instance, in considering the first character of the four-letter word group, representative "**ine**" corresponds to **"dine", "fine", "wine", "nine", "mine", "vine", "pine", "line".** Representative "**oot**" corresponds to **"boot", "foot", "hoot", "loot", "soot", "zoot".** Each individual `List` that is a value in this latest `Map` forms a clique of words in which any word can

be changed to any other word by a one- character substitution, so after this latest `Map` is constructed, it is easy to traverse it and add entries to the original `Map` that is being computed. We would then proceed to the second character of the four-letter word group, with a new `Map`. And then the third character, and finally the fourth character.

The general outline is given in Figure 5.

```
for each group g, containing words of length len
    for each position p (ranging from 0 to len-1)
    {
        Make an empty Map<String,List<String> > repsToWords
        for each word w
        {
            Obtain w's representative by removing position p
            Update repsToWords
        }
        Use cliques in repsToWords to update adjWords map
    }
```

Figure 5 The general algorithm for the third approach.

Figure 6 contains an implementation of this algorithm. The running time improves significantly. It is interesting to note that although the use of the additional `Maps` makes the algorithm faster, and the syntax is relatively clean, the code makes no use of the fact that the keys of the `Map` are maintained in sorted order.

As such, it is possible that a data structure that supports the `Map` operations but does not guarantee sorted order can perform better, since it is being asked to do less.

```
1       // Computes a map in which the keys are words and values are Lists of words
2       // that differ in only one character from the corresponding key.
3       // Uses an efficient algorithm that is O(N log N) with a TreeMap.
4       public static Map<String,List<String>>
5       computeAdjacentWords( List<String> words )
6       {
7           Map<String,List<String>> adjWords = new TreeMap<>( );
8           Map<Integer,List<String>> wordsByLength = new TreeMap<>( );
9
10            // Group the words by their length
11          for( String w : words )
12              update( wordsByLength, w.length( ), w );
13
14            // Work on each group separately
15          for( Map.Entry<Integer,List<String>> entry : wordsByLength.entrySet( ) )
16          {
17              List<String> groupsWords = entry.getValue( );
18              int groupNum = entry.getKey( );
19
20              // Work on each position in each group
21              for( int i = 0; i < groupNum; i++ )
22              {
23                  // Remove one character in specified position, computing
24                  // representative.  Words with same representative are
25                  // adjacent, so first populate a map ...
26                  Map<String,List<String>> repToWord = new TreeMap<>( );
27
28                  for( String str : groupsWords )
29                  {
30                      String rep = str.substring( 0, i ) + str.substring( i + 1 );
31                      update( repToWord, rep, str );
32                  }
33
34                  // and then look for map values with more than one string
35                  for( List<String> wordClique : repToWord.values( ) )
36                      if( wordClique.size( ) >= 2 )
37                          for( String s1 : wordClique )
38                              for( String s2 : wordClique )
39                                  if( s1 != s2 )
40                                      update( adjWords, s1, s2 );
41              }
42          }
43
44          return adjWords;
45      }
```

Figure 6 Function to compute a map containing words as keys and list of words that differ in only one letter as values (the third approach).