CS2302 - Data Structures
Fall 2017
Lab # 6
Word Clustering
Deadline: Friday, November 17, 11:59 p.m.

In the previous lab, you used a hash table to store and access the vector representations (embeddings) of a large set of English words. In this lab, you will use this table to solve a more complex NLP task. This task consists of creating clusters of similar words using a hash table, a heap, and a disjoint set forest. To accomplish this, you must write a program that does the following:

1. Populate your hash table using *glove.6B.50d.txt*, as done in Lab 5. You may want to use the following hash function.

```
private int h(String S){
  int h = 0;
  for(int i=0;i<S.length();i++)
    h = (h*27+S.charAt(i))%H.length;
  return h;
}
```

2. Read the words in file *Lab6Words.txt* provided in the class website to an array of strings.

3. Compute the similarity between every pair of words in the file by calculating the cosine distance between their vector representations. Every time you compute the similarity between two of words, create an instance of the *HeapNode* class (see below) and insert it into a heap. The idea is to build a heap, where the root is the node that contains the two most similar words in *Lab6Words.txt*.

   (a) Use the following *HeapNode* class definition:

```
public class HeapNode {
   public String word0;
   public String word1;
   public double similarity;

   public HeapNode(String word0, String word1, double similarity) {
      super();
      this.word0 = word0;
      this.word1 = word1;
      this.similarity = similarity;
   }
}
```

   (b) Use the following code fragment as the base for your *Heap* class.

```
public class Heap {

   /*
   In class, we used H[0] to store the size of the heap.
   In this lab, we will 'waste' H[0], and use the field 'size' to store the
   current size of the heap.
   */

   public HeapNode[] H;
   public int size;

   public Heap(int capacity) {
      H = new HeapNode[capacity + 1]; //We need to add 1 since we're 'wasting' H[0]
      this.size = 0;
   }

   public void insert(HeapNode node){
      //Use node.similarity * -1 when inserting 'node' into the heap.
      //We have to do this because our heap puts at the top the node with the
      //smallest value, and we actually want to achieve the opposite.
      //We want the words with the greatest similarity value to be at the top.
      //Because of this, you must use node.similarity * -1
      //as the node's value when inserting it into the heap.
      //...
   }

   public HeapNode extractMin() {
      //Use node.similarity * -1 in this method as well.
      //...
   }
}
```

4. After populating the heap, create an instance of the *DSF* class (see below). The main idea is to cluster very similar words by putting them together in the same set. Because of this, you need to have as many elements in your disjoint set forest as there are words in the array that contains the words in *Lab6Words.txt* file. Initially, every word will belong to its own set, meaning that each word will belong to a different cluster. Since each word is represented by a number in the DSF data structure, assume that its number representation is the position in which the word appears in the array. For example, element 0 in your DSF will represent the word *able*, element 1 will represent the word *algeria*, element 2 will represent the word *april*, etc. Use the following definition of the DSF class.

```
  /** Disjoint set forest class definition
  - Programmed by Olac Fuentes - Last modified October 20, 2017  **/
public class DSF{

   private int[] S;

   public DSF(int n){ /* Initialize disjoint set forest with n elements.
```

```
    Each element is a root */
       S = new int[n];
       for(int i=0;i<n;i++)
          S[i] = -1;
    }

    public int find_c(int i){ // Find with path compression
       if(S[i]<0)
          return i;

       return S[i] = find_c(S[i]);
    }

    public int union_c(int i, int j){ // Union with path compression
       int ri = find_c(i);
       int rj = find_c(j);

       if(ri == rj)
          return -1;
       S[rj] = ri;
       return 1;
    }

    public int union_by_size(int i, int j){
       int ri = find_c(i);
       int rj = find_c(j);
       if(ri==rj)return -1;
       if(S[ri]>S[rj]){
          S[rj]+=S[ri];
          S[ri]=rj;
       }
       else{
          S[ri]+=S[rj];
          S[rj]=ri;
       }
       return 1;
    }



    public void print(){
       for(int i=0;i<S.length;i++)
          System.out.print(S[i]+" ");
       System.out.println();
    }
}
```

5. After you have instantiated the *DSF* class, perform the *extractMin* operation on your heap n times (n

must be a variable in your program that you can easily tweak). After every *extractMin* operation, use the two words stored in the *HeapNode* object that *extractMin* returns to perform a union operation on your *DSF* object passing the numeric representation of the two words as parameters. For example, if *extractMin* returns a node that contains the words *spain* and *france*, you must call the union operation on your *DSF* object passing as arguments the numbers *289* and *97* since the words *spain* and *france* are represented by these two numbers, respectively.

6. Find and print the words contained in the m largest clusters (m must be a variable in your program that you can easily tweak). Hint: Use the unionBySize method when working with your *DSF* instance. In unionBySize, if i is a root, $S[i]$ = -(1 + number of elements in set), thus it's easy to find the largest set(s).

As usual, write a report describing your work. Play around with the variables n and m. Feel free to use a sub-set of the English words stored in the *Lab6Words.txt* file if your computer does not have enough resources to maintain and execute the operations of all the data structures that are used by your program.