Ruth Tilahun
CMSC 337
Lab 4
2/24/2024

Lab 4: Indexing

It is undeniable that searching on Google is very important to our society. In this lab we will see the underlying concepts in constructing an efficient and effective search engine which involves two main tasks: identification of the appropriate documents and ranking them based on their relevance.

Inverted index

In the initial phase we will create an inverted index – a mapping of words to their occurrences in a set of documents.

```
Given: F a set of documents IDs, one document per file

Return: A map containing an inverted index. --- Keys in the map
are words. Values are a list of instances of something like
WordDocumentLoc

Let: M : a map -- initially empty.

ForEach filename in F
    let wordloc = 0
    ForEach w in [the words in F]
        add to M for w a new record {filename, wordloc}
        set wordloc = wordloc + 1
return M
```

In the above pseudocode, we read a file line by line where each line contains a single word and create a *WordDocumentLoc* for each occurrence of a word and add it to a list in a map. *WordDocumentLoc* is a data structure we designed that contains the name of the file a word is found in and its location in that file in terms of the line number it is found at. We add the newly created *WordDocumentLoc* to a list in a map *M.* The keys in the map are the words, and the values are lists of *WordDocumentLoc* instances. If a word is not yet in the map, we create a new list. If a word is already in the map, the new *WordDocumentLoc* is added to the existing list. The result of this map, after reading all the files will be an inverted index where the index is a word that lets us access all the documents in which the word exists, including its location within each document.

Several issues might arise as the size of an inverted index increases. A larger inverted index requires more storage capacity and slows down the speed of indexing and data retrieval. To reduce the size of the inverted index we might exclude irrelevant words, words that are too common or words that are very rare. We can also group similar terms to reduce redundancy.

Using the inverted index

When searching for a word in the map, *M* we just constructed, the word might appear in the same file repeatedly. Therefore, we want to get the unique documents a word exists in and a separate list of all the locations of the word in each unique document. We will use two algorithms to accomplish this. The first one is to get all the unique documents containing the word we are looking for, and the second one to get all the locations of a word within each unique document.

Algorithm 1: Get unique documents for a set of words

This algorithm takes a set of words and returns a list that contains only the documents that contains the set of words. This is essential for doing a phrase lookup since we don't want documents that contain only some parts of the phrase.

```
Given: W a set of words
     : II an inverted index as above
Return: a list of document ids for documents containing all of
the words in W

   Let DI : a map from document ID to number (initially empty)
   ForEach w in W
      Let FW be the list of WordDocumentLoc objects for w from II
      Let UF = the unique Document IDs in FW

      ForEach uf in UF
          if uf is NOT in DI
             add uf to DI with a count of 1
          else
             update uf in DI by incrementing its value

Let R : a list of document IDs, initially empty
for docID, count in DI
    if count==len(W)
       add docID to R
return R
```

In this algorithm, the map, *DI* is used to store the count of how many times a document. For each word in the input array, *W*, we retrieve the corresponding list of *WordDocumentLoc* objects from the inverted index we created earlier. We use *FW* to store the list we retrieved. Then we populate the list *UF,* with the unique document IDs from the *FW* list using the following algorithm:

```
Given: FW a list of WordDocumentLoc objects from the inverted
index that a word is mapped to
Return:UF a list of unique document IDs in FW

   ForEach wordDocL in FW:
      Let docID be the document ID associated with wordDocL
      if UF doesn't contain docID
         add docID to UF
return UF
```

After getting list the unique document IDs, we then iterate over the list and update the *DI* map with the count of how many times each document ID appears. If a document ID is not in the map, it is added with a count of 1, otherwise the count is incremented. For example, for the first word that we process, all of the document IDs will have a count of 1. When we are processing the second and the rest of words we check whether or not a document ID associated with the word is already in the map *DI* then we proceed with adding the document ID with a count of 1 or incrementing the count. For example, consider *W has* three words and we have *DI* = {[Doc1, 3], [Doc2, 2], [Doc3, 1], [Doc4, 3]}, where the key is the document ID, and the value is count. This tells us that there are two documents, Doc1 and Doc4, in which all three of the words appear. Accordingly, only two and one of the three words appear in Doc2 and Doc3 respectively.
As stated earlier, when doing a phrase search, we are only interested in the documents that contain all the words in the phrase. To make a separate list for those document IDs, we will use the rest of the algorithm where we filter *DI* based on the number of counts. If *count* is equal to *len(W),* the number of words in *W,* we will add that document ID to a new list *R.* In the example above, *count* for Doc1 and Doc4 is 3, which equals to *len(W) = 3,* so we will add those document IDs to *R.*

Algorithm 2: Get locations of words

In this algorithm, given a word and a document ID obtained from Algorithm 1, we will create a list of the locations of the word in that document.

```
Given: word we are searching

     : docID document the word is found in
     : II an inverted index as above
Return: a list of locations of word in docID
Let locs be an empty list of integers
Let WL be the list of WordDocumentLoc instances for w from II
```

```
ForEach WordDocumentLoc instance in WL:
    if the document ID of the WordDocumentLoc instance == docID:
      add the location of the WordDocumentLoc instance to locs
return locs
```

The procedure iterates over a list of *WordDocumentLoc* instances for the word in the inverted index we created earlier and adds the word location to the list of *locs* if the document ID of the *WordDocumentLoc* instance matches the given document ID. We will do this for every word we are looking up. We will discuss why we need all the locations of the words we are searching for in the next section.

Finding the "best" match

If we consider the 'documents' we were processing in the above algorithms as web pages, so far, we have implemented the first of the two phases of web search which John MacCormick refers to as matching and ranking. (Maccormick, 2020)

Now, if we are performing a word or phrase search among, let's say a thousand web pages, our primary interest is unlikely to be getting a list of all the documents the word or phrase occurs in. Instead, we're probably only interested in a few of those files. Therefore, our search engine must pick the best few matches and display them in an order such that the 'better' documents are first. This process is called ranking. One way to define 'better' is: if the words we are looking for appear nearer to one another in a document then that document is more relevant. We use this logic because the set of words we are searching for will likely have more meaning if they are closer to each other. However, this isn't the only way to determine if a document is more relevant than another. Additionally, we may choose to ignore very common words such as "the" and "of" when ranking documents. These words not only are very common in that they appear in most, if not all, texts but they also occur frequently within a single document. This makes it harder to find relevant documents since the result of matching will be a very large collection of documents and locations of the words. In terms of efficiency, it will take a lot more space to keep track of all the documents these words are found in. The other reason is that these types of words don't hold significant meaning by themselves, thus they cannot truly be considered as key words.

In this lab, we consider a document more relevant based on the proximity between the words we are searching for. Using the list of locations that we obtained from Algorithm 2, we will find the closest pair of words in a document (Assuming that we only look up a pair of words).

Finding the closest pair of words

We will consider two ways of finding the closest pair of words in a document. The first one is a brute force algorithm where we compare each location of the first word with each location of the second word to find a pair of locations thar are closest to each other.

```
Given: W1: a list of the locations of word 1 in a document
       W2: a list of the locations of word 2 in the same document
Return: the smallest difference between a number in W1 and an
number in W2.

Let: d = abs(first item in W1 - first item in W2)
ForEach w1 in W1
    ForEach w2 in W2
        if abs(w1-w2) < d
           d = abs(w1-w2)
```

In this algorithm we compare each *w1 in W1* with each *w2 in W2* and update the smallest difference, *d.* If the absolute difference between the current elements *w1* and *w2* is smaller than the current smallest difference, we update *d* with the new, smallest difference. The time complexity of this algorithm is $\Theta(n^2)$.

A more efficient way to find the closest pair of words is as follows:

```
Given: W1: a list of the locations of word 1 in a document
       W2: a list of the locations of word 2 in the same document
Return: the smallest difference between a number in W1 and an
number in W2.

sort the contents of W1
sort the contents of W2

Let: idx1 = 0 // index in W1
     idx2 = 0 // index in W2
     len1 = length of W1
     len2 = length of W2
     d = abs(W1[idx1] - W2[idx2]) // distance between the first
items in W1 and W2
While idx1<len1 and idx2<len2
   if abs(W1[idx1]-W2[idx2]) < d
      d = abs(W1[idx1]-W2[idx2])
   if W1[idx1] < W2[idx2]
      idx1++
   else
      idx2++
```

The main improvement of this algorithm is that we have sorted the two lists so that we only iterate over each list only once. The loop continues as long as *idx1* and *idx2* are both within the bounds of W1 and W2, respectively. Inside the loop, we check if the absolute difference between the current elements *W1[idx1]* and *W2[idx2]* is smaller than the current smallest difference, and update *d* with the new, smallest difference accordingly. Then we increase the index which currently has the smaller location. To understand this better, let's look at the following example:

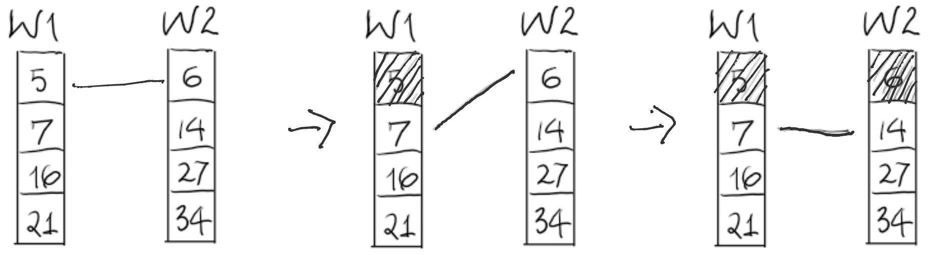

Figure 1: line between the numbers indicates which indices we are currently comparing. The numbers are locations of W1 and W2 in the same document

In the figure above, we have two sorted lists. Comparing the first elements of each list gives us a difference of 1. Thus, *d* is 1. The next step is choosing with index to increment. Based on our algorithm we increment the index at which there is the smaller location. Since the lists are sorted, we know that if 5 is less than 6 it is also less than everything that follows 6. Since everything after 6 is greater than 6, it follows that the shortest distance for 5 happens with 6. After incrementing the index of *W1*, we will compare 7 with 6 to get a distance of 1. Thus, the *d* remains the same. Since 6 is less than 7, we know we have found the shortest distance 6 can have across all the numbers in *W1,* so we increment the index of *W2,* and so on.

This algorithm has a time complexity of Θ(nlogn), if we are sorting the lists, or Θ(n) if the lists are already sorted.

After finding the closest pairs in each document, we will order the numbers such that the closest distance is displayed first. If we want to use this algorithm to find phrases, then we will only consider pairs at a distance of 1 to be relevant.

Summary

In this lab we have looked at one way to design a search engine. The first step is to find all documents that contain the word or phrase we are looking for, called matching. The second step is to determine an ordering of all the matches to have the most relevant documents appear first, known as ranking. We have determined proximity between words to be the measure of relevance and implemented an efficient way to calculate the shortest distance between a pair of words in a document.

References

Maccormick, J. (2020). *NINE ALGORITHMS THAT CHANGED THE FUTURE : the ingenious ideas that drive today's computers.* S.L.: Princeton University Pres.

Appendix A: Java Implementation

```
public static void main(String args[]) {
        String words[] = { "elizabeth", "emma"};

        //create a map of words to worddocumentloc
        for (int i = 0; i < args.length; i++) {
            invertedIndex(args[i]);
        }

        //get document IDs that contain both words
        ArrayList<String> docIdL = docIdList(words);

        ArrayList<CloserstPair> CP = new ArrayList<>();
        for (int i = 0; i < docIdL.size(); i++) {
            ArrayList<Integer> W1 = wordLocs(words[0],
docIdL.get(i));
            ArrayList<Integer> W2 = wordLocs(words[1],
docIdL.get(i));
            if(W1.size() != 0 && W2.size() != 0) {
                int closest = closestPair(W1, W2);
                CP.add(new CloserstPair(closest, docIdL.get(i)));
            }

        }
        //sort the list by closest pair (ascending)
        Collections.sort(CP);
        System.out.println("[" + words[0] + " " + words[1] + "]");
        for (int i = 1; i <= CP.size(); i++) {
            System.out.println(i + "\t" + CP.get(i - 1));
        }
    }

    public static ArrayList<Integer> wordLocs(String word, String
docID){
        ArrayList<Integer> locs = new ArrayList<>();
        ArrayList<WordDocumentLoc> worDocLocs = M.get(word);
        for(WordDocumentLoc wdl : worDocLocs){
            if(wdl.getDocumentId().equals(docID)){
                locs.add(wdl.getWordLoc());
            }
        }
        return locs;
    }
    public static ArrayList<String> docIdList(String[] words) {
        HashMap<String, Integer> DI = new HashMap<>();
        ArrayList<WordDocumentLoc> FW = new ArrayList<>();
        ArrayList<String> UF = new ArrayList<>(); // unique files /
document IDs
        for (int i = 0; i < words.length; i++) {
            String key = words[i];
            FW = (M.get(key));
            if (FW != null) {
                getUniqueIDs(FW, UF);
                for (int j = 0; j < UF.size(); j++) {
                    String docID = UF.get(j);
                    if (DI.get(docID) == null) {
                        DI.put(docID, 1);
                    } else {
                        DI.put(docID, DI.get(docID) + 1);
                    }
                }
            }
        }
        ArrayList<String> R = new ArrayList<>();
        for (String key : DI.keySet()) { // for each key in DI
            if (DI.get(key) == words.length) {
                R.add(key);
            }
        }

        return R;
    }

    public static void getUniqueIDs(ArrayList<WordDocumentLoc> FW,
ArrayList<String> UF) {
        for (int i = 0; i < FW.size(); i++) {
            if (!UF.contains(FW.get(i).getDocumentId())) {
                UF.add(FW.get(i).getDocumentId());
            }
        }
    }

    public static int closestPair(ArrayList<Integer> W1,
ArrayList<Integer> W2) {
        Collections.sort(W1);
        Collections.sort(W2);
        int idx1 = 0;
        int idx2 = 0;
        int len1 = W1.size();
        int len2 = W2.size();
        int minDiff = Math.abs(W1.get(idx1) - W2.get(idx2));

        while (idx1 < len1 && idx2 < len2) {
            int diff = Math.abs(W1.get(idx1) - W2.get(idx2));
            if (diff < minDiff) {
                minDiff = diff;
            }
            if (W1.get(idx1) < W2.get(idx2)) {
                idx1++;
            } else {
                idx2++;
            }
        }
        return minDiff;
    }

    public static void invertedIndex(String docID) {
        int wLoc = 1;
        try (Scanner input = new Scanner(new File(docID))) {
            while (input.hasNextLine()) {
                WordDocumentLoc newWord = new WordDocumentLoc(docID,
wLoc);
                String w = input.nextLine().toLowerCase();
                if (M.get(w) == null) {
                    ArrayList<WordDocumentLoc> locList = new
ArrayList<WordDocumentLoc>();
                    locList.add(newWord);
                    M.put(w, locList);
                } else
                    M.get(w).add(newWord);
                wLoc++;
            }

        } catch (Exception ee) {
            System.err.println(ee);

        }
    }
}
```

```
class WordDocumentLoc {
    private String documentId;
    private int wordLoc;

    public WordDocumentLoc(String docId, int wordL) {
        this.documentId = docId;
        this.wordLoc = wordL;
    }
    public String getDocumentId() {
        return this.documentId;
    }
    public int getWordLoc() {
        return this.wordLoc;
    }
    public String toString() {
        return "{" + this.documentId + " " + this.wordLoc + "}";
    }
}

class CloserstPair implements Comparable<CloserstPair>{
    int closestP;
    String docID;

    public CloserstPair(int cp, String dID) {
        this.closestP = cp;
        this.docID = dID;
    }
    public int getClosestP() {
        return this.closestP;
    }
    public String getDocID() {
        return this.docID;
    }
    public int compareTo(CloserstPair cp){
        if (getClosestP() > cp.getClosestP()) {
            return 1;
        } else if (getClosestP() == cp.getClosestP()) {
            return 0;
        } else
            return -1;
    }
    public String toString() {
        return  this.closestP + "\t" + this.docID;
    }
```

Appendix B: Results

```
[all good]
1       1       AustenOne_102.txx
2       1       GibonOne_539.txx
3       1       ScottOne_42.txx
4       1       AustenOne_180.txx
5       1       ScottOne_207.txx
6       1       AustenOne_7.txx
7       1       ScottOne_291.txx
8       1       AustenOne_374.txx
9       1       ScottOne_286.txx

[north heroism]
1       123     GibonOne_446.txx
2       306     ScottOne_352.txx
3       1395    AustenOne_78.txx
4       2388    GibonOne_288.txx
5       2771    GibonOne_326.txx
6       2990    GibonOne_267.txx

[mary contrary]
1       84      ScottOne_27.txx
2       148     AustenOne_182.txx
3       368     AustenOne_192.txx
4       441     ScottOne_207.txx
5       442     AustenOne_265.txx
6       511     ScottOne_158.txx
7       542     AustenOne_84.txx
8       557     AustenOne_61.txx
9       609     AustenOne_258.txx
10      678     AustenOne_87.txx
11      686     AustenOne_300.txx
12      693     AustenOne_191.txx
13      756     AustenOne_62.txx
14      918     AustenOne_69.txx
15      930     AustenOne_78.txx
16      980     AustenOne_92.txx
17      1254    ScottOne_177.txx
18      1257    ScottOne_260.txx
19      1377    AustenOne_181.txx
20      1442    ScottOne_29.txx
21      1857    ScottOne_277.txx
22      1969    ScottOne_26.txx
23      2029    AustenOne_271.txx
```

```
24      2202    AustenOne_391.txx
25      2362    AustenOne_189.txx
26      2453    AustenOne_81.txx
27      3355    GibonOne_467.txx
```