Search Engines Decoded:
Inverted Indexing, Relevancy, and Ranking

Raccoon Firefly

February 2024

1 Introduction and Brief Background

When searching on Google, I almost always click the first link that pops up, expecting it to be the best (most relevant, most polished, most credible, etc.).

Gravitating towards the first suggested links in a web browser intuitively makes sense, since we'd expect them to be the "better" links. However, computers do not have this intuitive expectation or innate knowledge of what makes a link "better." Programmers thus need to translate this human expectation into a set of more concrete criteria so that a computer can know not only which links may be relevant to a search, but also which links may be more relevant ("better") than the others.

Web page links can be evaluated for "betterness" by a wide variety of factors that ultimately depend on two main fields of consideration: a link's relationship with other links that reference it and a link's content. Search engines of the earlier internet tended to consider the first field by evaluating a link as "better" if there were a high number of hyperlinks that link back to it. As per the second field, the links that contain keywords relevant to a given search query were deemed as "better." As we discussed earlier, link "betterness" is a human perception, not a computer's one. So, as we refined our definition of "betterness" to include concepts like authority/credibility of a source, quality of link content, etc., newer algorithms (like PageRank) likewise evolved to translate these into code.

For this report, we will only be focusing on the second field of link evaluation: link content. This will help us gain a basic understanding of how search engines retrieve relevant links at a foundational level. We will begin by examining an inverted indexing algorithm that mirrors what search engines may use to look for queried words across multiple links. Then, we will apply a simple ranking algorithm to mimic a more digestible form of determining link "betterness." Along the way, we'll evaluate how effective these approaches are, especially when applied to the larger scope of in-practice search engines like Google.

2 Getting into the Details

The scope of this report requires that we think of link content as its purely textual core. Similarly, we can reduce this idea of the Internet (with its many links and web pages) to simply a large collection of documents. In this context, searching for a link by a keyword search query (i.e. searching for a chocolate cake recipe by typing "chocolate cake recipe" into Google), is thus a problem of finding words (or sets of words/phrases) in a large collection of documents.

2.1 Predicting Relevancy Using Inverted Indexing

Under the premise that a relevant link is one that contains a set of given keywords (i.e "chocolate," "cake," and "recipe"), we can construct an inverted index to streamline keyword lookup across a large collection of documents.

An inverted index is a data structure that maps words to the documents that they appear in (and, often, their locations in their respective documents). Constructing an inverted index requires that we parse each document, extract unique words, and update the index with the found location information.

This report assumes that each document is organized so that each line contains one word. This way, we can use line numbers to represent a word's location in a document. Keeping this in mind, a simple algorithm for building an inverted index could look like the following:

```
Given: D a set of documents
Return: An inverted index map (keys: words, values: list of locations)
Initialize an empty map M (word to location list)
for each document in D
    let location = 0
    for each word in D
        if word is not already a key in M
            add word to M with an initially empty list for its value
        add location to the list of locations for this word
        location = location + 1
return M
```

Figure 1: An algorithm to build an inverted index given a set of documents. Takes the form of a map in which keys are words and values are lists of locations at which the key word occurs. Based on algorithms described in Chapter 2 of *Nine Algorithms That Changed the Future* (MacCormack, 2012).

In Figure 1, we take a set of documents, parse each one, and add unique words and their locations (line numbers) in the document to an inverted index map M. However, the algorithm lacks a way to differentiate word locations from one document to another.

To address this ambiguity, for each instance of a word, we can store the name of the document it's in (or a link's URL if we think in terms of the broader search engine scope) in addition to its location/line number in the document. Code (in Go) to store this information may look like so:

```
type WordDocLoc struct {
    docId string // filename of file where word is found
    wordLoc int // line number where word is found
}
```

Utilizing the algorithm depicted in Figure 1 with the 'WordDocLoc' data type or an analogous one instead of solely relying on a line number for location will allow us to pinpoint the exact document and location of a particular word instance within the map.

After parsing the entire set of documents and mapping each unique word to its list of specific locations, we're left with an inverted index map. With an inverted index already built, we no longer have to scan all the documents to see if/where a given word appears. This allows us to perform repeated lookups quickly. However, this lookup speed comes with a cost. Depending on implementation details, this data structure can take up a varying amount of memory, yet in most cases, this amount is very large.

Let's look to the Go code behind this report to generate an example memory approximation:

```
Given that:
    All documents are stored in one directory.
    Each document is organized one word per line.
    Results from bash 'wc' command:
        - Total number of words: approximately 3,000,000 (non unique)
        (can also be thought of as the total num of word instances)
    A WordDocLoc struct contains a string and an int.
        - The 'string' data type contains a pointer to the underlying
        char array (8 bytes) and its length (8 bytes) for a total of
        16 bytes
        - The 'int' data type (on a 64-bit) machine takes up 8 bytes
        - A WordDocLoc instance thus takes up a total of 24 bytes
```

```
        The values of the map are slices of WordDocLocs.
            - A slice is made up of a pointer (8 bytes) and two integers
            to store a length and a capacity (8 bytes each) for a total
            of 24 bytes
        A Go map is a pointer, and will take up 8 bytes.
        There are approximately 70,000 unique words in the map (len(map)).
        We assume an average of 5 bytes per word (word length of 5 characters).


    We'll need:
        - 3,000,000 word instances * the 24 bytes for a WordDocLoc instance
        = 72,000,000 bytes to account for all word instance information
        - 70,000 unique words * 16 bytes string overhead * 5 bytes for the
        average word = 5,600,000 bytes to account for all keys in the map
        - 70,000 unique words * 24 bytes for a slice to store a word's instances
        = 1,680,000 bytes to account for all the values in the map

        Thus, to store just the content of the map, we need approximately:
        72,000,000 + 5,600,000 + 1,680,000 = 79,280,000 bytes

Ultimately, the 79,280,000 bytes of content + the 8 bytes for the map pointer
will take up a rough estimate of 80,000,000 bytes, which is about 80 MB.
```

Figure 2: Computation of the amount of memory taken up by the built inverted index previously described. The inverted index is expected to take up approximately **80 MB** when implemented in Go.

Figure 2 estimates that an implementation of the Figure 1 algorithm to build an inverted index will result in a data structure of about 80 MB. (In more human-understandable terms, that's about 80 minutes of MP3 audio.) 80 MB is a fairly large chunk of memory, and when compared to the 44 MB of data that our set of documents takes[1], the size of the inverted index only seems heftier. The amount of memory needed nearly doubles.

2.2 Evaluating Memory Requirements of Inverted Indexing

Keep in mind that the 80 MB of memory we calculated earlier is specific to the 44 MB set of documents that this report is concerned with. Actual search engines like Google need to handle over 100,000 TB of data Google, n.d. Indexing data of sizes on this order can be costly to maintain. First of all, storing all this data

[1]Dataset can be found at https://www.cs.brynmawr.edu/cs337/Lab04Data/

requires much more space and infrastructure costs. Additionally, having an index this massive would take a toll on search performance, forcing users to wait longer for their search results. Updating the index to process new web pages would be a similarly painstaking process.

To try and remediate some of the issues that come up when dealing with a large index, companies like Google can divvy up their index across multiple servers. This helps promote easier scalability, reduces localized errors from shutting down the entire search engine, and may be less costly to manage.

Moreover, while the inverted index for this report takes up almost double the amount of space than its raw data, in practice, inverted indices for the search engines of today typically take up less space than the raw data they index. Various compression techniques Kumar et al., 2014 can be applied to the inverted index to reduce size, but they fall out of the scope of this report.

However, an easily digestible way to reduce index size is to simply filter out common filler words (i.e. "the," "of," "is," "and," etc.). These words, also known as stop words, are ubiquitous, so their list of instances and locations would be extensive. Furthermore, stop words don't provide that much meaning, so processing them and storing their locations is not useful and/or necessary. Thus, ignoring stop words entirely when indexing could free up a lot of space.

Similarly, super rare/low-frequency terms can be ignored to save space. However, these words are more likely to hold more significant meaning than stop words, so disregarding them is not entirely trivial. Queries that target these low-frequency terms may be rare, but they may still occur. Users should be able to search with specificity and focused intent.

Another way to reduce index size by word filtering, which is arguably safer than ignoring rare terms, is to treat words with the same stem as the same word when creating a key in the inverted index map. For instance, the words "listen," "listening," and "listened" share the same stem "listen," so all their instances can be mapped to the stem word to avoid redundant map entries. This approach would also free up index memory space.

2.3 Using an Inverted Index

Regardless of size, after building an inverted index, we can leverage it to quickly process user queries.

Processing a one-word query simply requires a lookup of the associated key in the inverted index. One would only need to ask the inverted index map for the word to get its associated list of documents and document locations (line numbers). The documents in the retrieved list could then just be filtered out for duplicates in order to give the user a set of unique documents to look through. The same applies in the

broader scope of search engine lookup if we think in terms of links/web pages in place of documents.

Processing multiple-word queries gets a little more complex, but nevertheless remains effective. To do so, one would need to retrieve the set of unique documents that contain each word in the query to ultimately return the intersection of that set. An algorithm to perform this task could look like so:

```
    Given: W a set of words
            I an inverted index
    Return: the set of unique documents (document names) containing all
words in W
    let D = a map from documents to number
    for each w in W
        let wD = the set of documents that contain w (retrieved from I)
        let uD = the set of unique documents in wD

        for each unique document u in uD
            if u is NOT in D
                add u to D with count of 1
            else
                increment the count of u in D

    let R = the set of unique documents containing all words in W
    for d in D
        if count == len(W) // if all words in W appear in d
            add d to R
    return R
```

Figure 3: An algorithm to process multiple-word queries. Looks at each word, finds the unique documents that contain said word, and then finds the intersection of documents that contain all words in the query.

Figure 3 above showcases that in order to process multiple-word queries, one must first process each word in the query, find the list of documents that contain each word, and then from there collect the common set of documents. The algorithm utilizes a second map D from document-to-count across all words in the query to avoid repeatedly scanning for common documents. For this reason, the second map D is useful and efficient, but it does take up a small amount of extra memory.

2.4 Ranking Based on "Betterness"

The algorithm described in Figure 3 gives us a set of documents that contain all words in a multiple-word query. In terms of querying for web page links in a search engine like Google, the algorithm in Figure 3 would spit out an arbitrarily organized list of all web pages that contain the given keywords. As users, this arbitrary list is not as useful to us as a ranked list. In other words, we want "better" web page links to show up first, so we have less sifting to do on our own.

To mimic this concept of ranking our query results by "betterness," we first have to define what "better" means for our specific problem. The code behind this report considers "better" documents as ones in which the distance between given queried words is minimal. If the queried words are closer in location in a given document, that document should be ranked as "better" and appear higher in the returned list.

Consider the comparison of two documents below:

document1.txt	document2.txt
the **quick** brown **fox** jumps over the lazy dog	the over dog the lazy brown jumps **quick** **fox**

Figure 4: Table showcasing two example documents to highlight how proximity-based ranking works. The document in which queried words appear closer together gets priority.

If a user queried for "quick fox," Document 2 in Figure 4 would be the "better" document because the words "quick" and "fox" are only 1 line apart as compared to the 2 line difference in Document 1. Document 2 should thus appear before Document 1 in the list.

Based on this definition of "betterness," Figure 5 below provides an example of an algorithm to find the minimum distance between queried words in a document. Please note that for now, this algorithm only considers two-word queries. Future work to extend this solution to support queries larger than two words would be useful to explore, but it is out of the scope of this specific report.

```
Given: W1 a list of the locations (line nums) of word 1 in a document
    W2 a list of the locations (line nums) of word 2 in the same document
Return: the smallest location difference between word 1 and word 2
    (a.k.a. the min difference between a num in W1 and a num in W2)

sort the contents of W1
sort the contents of W2

let i1 = 0 // index in W1
let i2 = 0 // index in W2
let len1 = length of W1
let len2 = length of W2
let d = abs(W1[i1] - W2[i2]) // distance between 1st items in the lists

while i1 < len1 and i2 < len2
    if abs(W1[i1] - W2[i2]) < d
        d = abs(W1[i1] - W2[i2])
    if W1[i1] < W2[i2]
        i1++
    else
        i2++
return d
```

Figure 5: An algorithm to compute the minimum distance between two given words in a given document. Used to determine a given documents "betterness" score.

The algorithm described in Figure 5 iterates through both lists W1 and W2 simultaneously, saving time when computing and comparing the absolute difference between the elements in W1 and the elements in W2. The smallest distance is tracked by the variable d, which will get returned once one of the lists is fully traversed. The simultaneous iteration is applicable because of the previous sorting of both lists W1 and W2. Because both lists are sorted in ascending order, once the algorithm finds an element in W1 (at i1) that is smaller than an element at the same index (i2) in W2, it knows that since the next element in W1 (at i1++) is larger, the distance between the next element in W1 and the element at the same index (i2) in W2 will be smaller. Essentially, the sorting of both lists W1 and W2 gives the algorithm predictably ordered lists in which known assumptions can be made to avoid redundant comparisons.

Figure 5's algorithm can be used to sort a set of documents that contain all words in a multiple-word query generated by the algorithm in Figure 3. Considering only two-word queries, once Figure 3's algorithm provides the documents that contain

both queried words, an algorithm to sort the provided documents that uses Figure 5's algorithm could look like so:

```
Given: word1 the first queried word
        word2 the second queried word
        D the set of documents containing both words word1 and word2
        I the inverted index
Return: a map of the given documents to their minimum distance between
instances of word1and word2 that is ordered based on the minimum distance

let M = map of documents to minimum distance
sort the given documents based on the following comparison function:
    for each document x and y
        compute the smallest distance between word1 and word2 in
            document x using the algorithm described in Figure 5
        compute the smallest distance between word1 and word2 in
            document y using the algorithm described in Figure 5
        store the computed distances in M
        compare the computed distances of x and y
            if the distance of x is smaller than that of y
                return true
            else
                return false
return M (now sorted)
```

Figure 6: An algorithm to sort documents containing both given keywords by their "betterness" ranking determined by the proximity-prioritizing algorithm described in Figure 5. Returns a map of relevant documents that is ordered with "better" documents appearing first.

Figure 6 showcases an algorithm that takes the set of documents provided by Figure 3 and reorders them based on the definition of "betterness" as defined by Figure 5. For clarity for this report, as well as clarity for debugging purposes, the map resulting from Figure 6 stores the documents as well as their minimum distance. This, however, is not necessary to the problem at large. To save space, one could just return the ordered set of documents in a simpler list data type structure (ex: array, array list, slice, etc.).

Ultimately, using Figure 6's algorithm alongside Figure 5's approach provides users with a list of documents containing their queried words, prioritizing those where the words are closer together. However, prioritizing documents in which query words are closer together does not always directly translate to the document being "better" in a user's eyes.

Consider the following example:

```
The user queries the two words ‘‘turkeys eat’’ with the intent to find
common foods that turkeys eat.
```

The algorithms described above would only provide the user with a set of documents that contain both the words “turkeys” and “eat,” with the first document being the one in which a found instance of the word “turkeys” is relatively close to a found instance of the word “eat.” Depending on the document content, the suggested/first document given to the user might not actually be what the user is looking for. For instance, the words “turkeys” and “eat” might appear close to each other in an irrelevant blog post describing a meal: “My family eats two turkeys during Thanksgiving. The two words might even appear next to each other, but in an order that doesn’t make sense for the user’s intention: “Bobcats eat turkeys.”

To accommodate scenarios like the one described above, we can adapt the algorithms in Figures 5 and 6 accordingly. To minimize recommending irrelevant documents to users that specifically want to find instances of their queried words as a phrase (“turkeys eat” rather than “turkeys” and “eat”), Figure 6’s algorithm can be adapted to filter out documents whose minimum distances are greater than 1. Additionally, to avoid prioritizing documents in which “eat turkeys” appear over documents in which “turkeys eat” (which is what the user wants), instead of relying on absolute differences, Figure 5’s algorithm could look at only positive differences (where word 1’s line number - word 2’s line number is greater than 0). (“turkeys eat” would give a difference of 1, while “eat turkeys” would give a distance of -1).

Along this line of thinking, our overall way of ranking on this proximity-prioritized definition of “betterness” can be further refined.

2.5 Refining the Definition of “Betterness”

The algorithms described previously rank documents relevant to a user’s query based on keyword proximity. While keyword proximity may be a useful parameter for judging document “betterness,” it’s not always conclusive. Keyword proximity should not be the sole deciding factor in determining “betterness.” Rather, we should rely on a variety of factors.

Because the scope of this report is looking specifically at this query-to-document problem (rather than the broader scoped, search engine query-to-web-page problem), we’ll focus on what deciding intrinsic factors we can use to refine our definition of “betterness”.

For instance, without relying on information external to the documents in question, we can consider the frequencies of queried keywords when ranking. Documents

in which user-queried words appear more frequently may contain more information that the user is looking for. Subsequently, a user might benefit from a ranking in which documents with high keyword frequencies are pushed to the top of the list. This can easily be achieved with the previously built inverted index, as that map already keeps track of all instances of a given keyword across all documents.

When factoring in keyword frequency when defining "betterness," an unintentional bias toward longer documents may appear. To minimize this bias, we can look at keyword frequency relative to document length. Normalizing keyword frequencies in this way prevents prioritizing longer documents simply because they are long.

Another intrinsic factor that can be considered is document recency. Documents with higher temporal relevance could be seen as "better" since they can provide users with the most up-to-date information. However, scraping (and updating) document time stamps would require extra space and time.

Though it falls outside the scope of this report, semantic similarity is another factor worth looking into when refining "betterness." Looking for words similar to queried words (ex: synonyms, related terms, etc.) will broaden a user's query to a larger set of documents that may be just as useful as (or more useful than) documents that only contain the given keywords.

It should be no surprise that when ranking suggested web pages, modern-day search engines consider the factors described above in addition to factors that fall out of the scope of this report. After reading this report, if you want to delve into the more complex ways of refining "betterness," exploring semantic similarity, contextual analysis, user behavior, popularity, and link authority may be good places to start.

3 Summary

Overall, this report explores the application of inverted indexing in keyword lookup across a collection of documents as a more digestible stand-in for the foundational concepts behind search engines. Beginning with a debrief on the basics of inverted indices, we've looked at an algorithm to construct an inverted index and examined its memory requirements. We've discussed that although inverted indices can be very large, actions like word filtering can help ease some memory concerns. Throughout this report, we've also dissected a ranking algorithm for a keyword proximity-based definition of "betterness" to gain a more solid understanding of how query results may be reordered to prioritize more preferable documents. Additionally, we've considered ways to refine the notion of document "betterness" by integrating intrinsic factors such as keyword frequency. All in all, by unraveling the core dynamics of document search and ranking at this level, this report demystifies the core parts of the search engines we see today.

References

Google. (n.d.). *How google search organizes information*. https://www.google.com/search/howsearchworks/how-search-works/organizing-information/#:~:text=The%20Google%20Search%20index%20contains,on%20every%20webpage%20we%20index

Kumar, C., Ramachander, N., & Nagineni, S. (2014). Decreasing inverted index size by using highly optimized compression scheme. *International Journal of Network Security*, *2*, 206–211.

MacCormack, J. (2012). *Nine algorithms that changed the future*. Princeton University Press.

Appendices

Appendix A

Source code for the implementations of the algorithms and program described above

```
package main

import (
    "fmt"
    "net/http"
    "bufio"
    "log"
    "strconv"
    "strings"
    "sort"
    "math"
)

const NUM_GIBON_FILES int = 565
const NUM_SCOTT_FILES int = 393
const NUM_AUSTEN_FILES int = 414
const FILE_SUFFIX string = "One_"
const FILE_TYPE string = ".txx"
const SOURCE_URL string = "https://www.cs.brynmawr.edu/cs337/Lab04Data/"
const WORDS_START int = 3 // skip first 2 lines (contains chapter and chapter num)

type WordDocLoc struct {
    docId string // filename of file where word is found
    wordLoc int // line number where word is found
}

func (w WordDocLoc) String() string {
    return fmt.Sprintf("{%s %d}", w.docId, w.wordLoc)
}

/* reads the file that corresponds to the given file info
 * and returns the document id and a string slice of lines
 */
func readFile(author string, fileNum int) (string, []string) {
    docId := author + FILE_SUFFIX + strconv.Itoa(fileNum) + FILE_TYPE
resp, httpErr := http.Get(SOURCE_URL + docId)

    if httpErr != nil {
        log.Fatal(httpErr)
    }

    defer resp.Body.Close()

    var lines []string
    scanner := bufio.NewScanner(resp.Body)

    for scanner.Scan() {
        lines = append(lines, scanner.Text())
    }

    scannerErr := scanner.Err()
    if scannerErr != nil {
        log.Fatal(scannerErr)
```

```
    }

    return docId, lines
}

/* given that each line contains a single word, parse the given slice of lines
 * and populate a map of each word to its list of wordDocLocs (file and line num)
*/
func parseFile(docId string, lines []string, wordMap map[string][]WordDocLoc) {
    wordLoc := WORDS_START
    for i := 2; i < len(lines); i++ {
        word := strings.ToLower(lines[i])
        val, valExists := wordMap[word];
        if !valExists {
            wordMap[word] = make([]WordDocLoc, 0) // initialize a new slice w/ default size 0
        }
        // add word's wordDocLoc to its value slice
        wordMap[word] = append(val, WordDocLoc{docId: docId, wordLoc: wordLoc})
        wordLoc += 1
    }
}

// find and return a list of files that contain the entire set of given words
func findFileContaining(words []string, wordMap map[string][]WordDocLoc) []string {
    // initialize a map of docIds to how many words from words slice they contain
    docToWordCount := make(map[string]int)
    for _, word := range words { // for each of the given set of words
        wordDocLocs := wordMap[word] // get its list wordDocLocs
        // find the unique doc ids in the wordDocLocs
        uniqueDocIds := make(map[string]bool)
        for _, wordDocLoc := range wordDocLocs {
            docId := wordDocLoc.docId
            if _, valExists := uniqueDocIds[docId]; !valExists {
                uniqueDocIds[docId] = true
            }
        }

        // for each unique doc id found, increment its count in the map
        for uniqueDocId, _ := range uniqueDocIds {
            if _, valExists := docToWordCount[uniqueDocId]; !valExists {
                docToWordCount[uniqueDocId] = 0
            }
            docToWordCount[uniqueDocId]++
        }
    }

    var files []string // initialize list of files that contain the entire set of words
    for docId, wordCount := range docToWordCount { //   for each docId in the map
        if wordCount == len(words) { // if a docId is associated with as many words as in the set
            files = append(files, docId) // add to list of files to return
        }
    }

    return files
}

/* reorders the given list of files by ranking,
 * where "better" files are put first (a "better" file can be defined as a file in which the
 * set of words it is meant to contain are located closer together; for now assuming 2 words)
 * returns a map of file to the distance between the two given words; map is not sorted
```

```
 */
func rankFilesContaining(files []string, word1 string, word2 string, wordMap map[string][]WordDocLoc) map[string]float6
    fileToDistance := make(map[string]float64)

    sort.Slice(files, func(x int, y int) bool {
        smallestDistance1 := findSmallestDistanceBetween(word1, word2, files[x], wordMap)
        smallestDistance2 := findSmallestDistanceBetween(word1, word2, files[y], wordMap)
        fileToDistance[files[x]] = smallestDistance1
        fileToDistance[files[y]] = smallestDistance2
        return smallestDistance1 < smallestDistance2
    })

    return fileToDistance
}

// finds the smallest distance between 2 given words in a given file
func findSmallestDistanceBetween(word1 string, word2 string, file string, wordMap map[string][]WordDocLoc) float64 {
    wordDocLocs1, _ := wordMap[word1]
    var word1LocsInFile []int // list of locations of word1 in the given file
    for _, wordDocLoc := range wordDocLocs1 {
        if wordDocLoc.docId == file {
            word1LocsInFile = append(word1LocsInFile, wordDocLoc.wordLoc)
        }
    }
    sort.Slice(word1LocsInFile, func(x int, y int) bool {
        return word1LocsInFile[x] < word1LocsInFile[y]
    })

    wordDocLocs2, _ := wordMap[word2]
    var word2LocsInFile []int // list of locations of word2 in the given file
    for _, wordDocLoc := range wordDocLocs2 {
        if wordDocLoc.docId == file {
            word2LocsInFile = append(word2LocsInFile, wordDocLoc.wordLoc)
        }
    }
    sort.Slice(word2LocsInFile, func(x int, y int) bool {
        return word2LocsInFile[x] < word2LocsInFile[y]
    })

    i1 := 0 // index in word1LocsInFile
    i2 := 0 // index in word2LocsInFile
    smallestDistance := math.Abs(float64(word1LocsInFile[i1]) - float64(word2LocsInFile[i2])) // distance between the f

    for i1 < len(word1LocsInFile) && i2 < len(word2LocsInFile) {
        distance := math.Abs(float64(word1LocsInFile[i1]) - float64(word2LocsInFile[i2]))
        if distance < smallestDistance {
            smallestDistance = distance
        }

        if word1LocsInFile[i1] < word2LocsInFile[i2] {
            i1 += 1
        } else {
            i2 += 1
        }
    }

    return smallestDistance
}

// prints a map of ranked files given list of files in correct order
```

```
func printRankedFiles(files []string, rankedFiles map[string]float64) {
    for i, file := range files {
        fmt.Printf("%d %6d %s\n", i+1, int(rankedFiles[file]), file)
    }
}

func main() {
    wordMap := make(map[string][]WordDocLoc)

    // read and parse all gibbon files
    for i := 1; i <= NUM_GIBON_FILES; i++ {
        docId, lines := readFile("Gibon", i)
        parseFile(docId, lines, wordMap)
    }

    // read and parse all scott files
    for i := 1; i <= NUM_SCOTT_FILES; i++ {
        docId, lines := readFile("Scott", i)
        parseFile(docId, lines, wordMap)
    }

    // read and parse all austen files
    for i := 1; i <= NUM_AUSTEN_FILES; i++ {
        docId, lines := readFile("Austen", i)
        parseFile(docId, lines, wordMap)
    }

    fmt.Printf("MAP SIZE IS %d\n", len(wordMap))

    // test searching files for pairs gets correct files, correctly ranked by distance
    pair1 := []string{"elizabeth", "emma"}
    pair2 := []string{"roy", "clan"}
    pair3 := []string{"legend", "legion"}

    files1 := findFileContaining(pair1, wordMap)
    files2 := findFileContaining(pair2, wordMap)
    files3 := findFileContaining(pair3, wordMap)

    fmt.Println(pair1)
    printRankedFiles(files1, rankFilesContaining(files1, pair1[0], pair1[1], wordMap))
    fmt.Printf("\n%v\n", pair2)
    printRankedFiles(files2, rankFilesContaining(files2, pair2[0], pair2[1], wordMap))
    fmt.Printf("\n%v\n", pair3)
    printRankedFiles(files3, rankFilesContaining(files3, pair3[0], pair3[1], wordMap))

    // second set of tests
    pair4 := []string{"all", "good"}
    pair5 := []string{"north", "heroism"}
    pair6 := []string{"mary", "contrary"}

    files4 := findFileContaining(pair4, wordMap)
    files5 := findFileContaining(pair5, wordMap)
    files6 := findFileContaining(pair6, wordMap)

    fmt.Println("\nAPPENDIX RESULTS:")
    fmt.Printf("\n%v\n", pair4)
    rankedFiles := rankFilesContaining(files4, pair4[0], pair4[1], wordMap)
    // only print files with distance 1
    for i, file := range files4 {
        if int(rankedFiles[file]) == 1 {
```

```
            fmt.Printf("%d %6d %s\n", i+1, int(rankedFiles[file]), file)
        }
    }

    fmt.Printf("\n%v\n", pair5)
    printRankedFiles(files5, rankFilesContaining(files5, pair5[0], pair5[1], wordMap))
    fmt.Printf("\n%v\n", pair6)
    printRankedFiles(files6, rankFilesContaining(files6, pair6[0], pair6[1], wordMap))

}
```

3.1 Appendix B

Results for given test queries when running the source code from Appendix A

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