Jordan Fox
CS 311
Programming Assignment 1 Report
Recitation 6

General discussion: In this project I made heavy use of concurrently implemented arraylists and hashtables. By having an arraylist of type E and a hashtable of type <E,Integer> items can be stored in both data structures, and be searched for by either index or value in O(1) time.

Crawl
Runtime: O(m+n)
Discussion:
Graph is implemented with an adjacency list for edges, vertices are stored with both an arraylist of type string where the string is the url, as well as a hashtable of type <String,Integer> where the string is the url and the integer is the index of that url in the array matrix. By doing this, all operations are O(1). The BFS implementation in crawl is near identical to that in the textbook with the following exceptions: graph implementation checks if it is full when adding a virtex, search stops when it has reached the max depth, and when adding edges, the graph checks if that edge already exists, since a link may exist multiple times in a page, this is an O(1) operation. Thus the runtime is that of a traditional BFS which is O(m+n).
Pseudocode:
Initialize graph web with seed and maxVertices
Initialize layer counter i to 0
Initialize layer L[i] to contain seed
while(L[i] is not empty and i <= maxDepth){
Initialize layer L[i+1]
For each url s in L[i]{
For all valid links from s to v {
If web does not contain v, and i != maxDepth, and web is not full{
Add v to web
Add v to L[i+1]
}
If web does not contain edge (s,v), add it
}
}
Increment i by 1
}
Return web

makeIndex
Runtime: O(n)
Discussion: For each page, two loops occur sequentially. The first parses the page and creates a tally (hashtable) of how many times each word occurs in that page, this process has runtime

that scales with the number of words in that page. The second is where values are scored in the inverted index, this scales with the number of unique words (arraylist) in the page, which in the worst case has runtime equal to that of the first loop.
Pseudocode:
Initialize hashtable invIndex of type <String,ArrayList<TaggedVertex<String>>>
For each page p{
Initialize hashtable of type <String, Integer> tally
Initialize arraylist of type <String> words
For each valid word w in p{
If tally contains key w{
Increment value of tally[w] by 1
} else {
Put value 1 in tally with key w
Add w to words
}
}
For each word d in words{
If invIndex does not contain key d, initialize empty list at invIndex[d]
Add (p,tally[w]) to list stored in invIndex[w]
}
}

Search
Runtime: O(nlogn)
Discussion: This algorithm works in two phases, the first calculates the r value for each result. This works in O(n) time. The second phase is performing merge sort on this created list. This is of course O(nlogn). The index class has a hashmap to find the index and thus indegree of a link when given the url in O(1) time. In pseudocode below, this is represented as id[v] to indicate the indegree of the page url v. Also for each tuple, v represents the url and t represents the number of occurrences of the word w.
Pseudocode:
If inverted index contains word w{
Initialize empty arraylist a
For each tuple (v,t) stored in the list at invIndex[w] {
add (v,t*id[v]) to a
}
Return merge sorted a
} else {
Return empty arraylist
}

SearchWithAnd
Runtime: O(nlogn)

Discussion: First, finds all pages containing w1 and stores that r value in a hashmap. Then finds all pages containing w2. When finding pages containing w2, pages that are contained as keys in the hashmap are stored in an arraylist. After this, that arraylist is sorted.
Pseudocode:
Initialize hashmap q1 of type <String,Integer>
For all tuples (v,t) in invIndex[w1] {
Put (v,t*id[v]) into q1
}
Initialize arraylist a
For tuples (v,t) in invIndex[w2]{
If q1 contains key v {
Add (v,t*id[v]+q1[v]) to a
}
}
Mergesort a and return

SearchWithOr
Runtime: O(nlogn)
Discussion: This uses a concurrently implemented hashtable and arraylist to add pages to the list during the w1 search and then find them later in O(1) time to use in the w2 search.
Pseudocode:
Initialize index counter i = 0
Initialize empty hashtable mapping of type <String,Integer>
Initialize empty arraylist a of type <TaggedVertex<String>>
For all pages containing w1{
Calculate r and add to a at index i
Add (p,i) to mapping
Increment i by 1
}
For all pages containing w2{
If mapping contains w2 {
Add r(w2) to r value in a
} else {
Calculate r and add it to a at index i
Add (p,i) to mapping
Increment i by 1
}
}
Mergesort a and return

searchWithNot

Runtime: O(nlogn)

Discussion: almost identical to searchWithAnd but with a different conditional so to not add pages containing both words
Pseudocode:
Initialize hashtable mapping of type <String,integer>
Initialize arraylist a of type TaggedVertex<String>
For all pages p2 containing w2{
Add (p,1) to mapping
}
For all pages p1 contiang w1{
If mapping does not contain key p1{
Calculate r and add to a
}
}
Mergesort a and return