# 311 Programing Assignment 1 Report

Zach DeMaris, James Taylor

# Pseudocode & Runtimes

Notes:

For Runtimes we call

vertices V and edge E

pages n and words m

tV = Tagged Vertex

TVWC = Tagged Vertex Word Count

(Coupling of TaggedVertex and an integer for word count, see justification of data structures for more)

### crawl

```
crawl() {
New Graph<String> webGraph
New Queue queue

queue.add (seedURL)
queue.add(null value to indicate end of layer)

while(!queue.isEmpty() && currentDepth <= maxDepth && currentPages
<= maxPages){

  URL = queue.pop()

  if(URL == null){
     Signals end of layer so increment layer count
     currentLayer++
     Continue to next URL
}


  For every adjacent URL {

    if(link should not be ignored){
     if(webGraph.hasEdge(URL, adjURL)) continue;

     if(webGraph.hasVertex(URL, adjURL)){
      webGraph.addEdge(URL, adjURL)
     }elseif(currentDepth < maxDepth && currentPages < maxPages){
```

```
      queue.add(adjURL)
      webGraph.addEdge(URL, adjURL)
      currentPages++;
    }

   }
  }
}
  return webGraph
}
```

## crawl runs in time O(V+E)

breakdown of steps and runtime:

This basic structure of crawl is BFS. Which runs in O(V + E). The public methods from graph used in this algorithm are addEdge, hasEdge, and hasVertex. All of these operations are performed in O(1) time because of the way we have implemented the Graph interface. See information about TwoWayGraph.java below. In addition to these calls, we also use the Jsoup library which are assumed to run in O(1). Therefore, our modified BFS that implements crawl still runs in O(V + E).

## makeIndex

```
makeIndex(){
 New Hashmap <String, ArrayList <TaggedVertexWordCount>> map
 New Hashmap <String, Integer> helperMap

 For (every webpage as a taggedVertex 'tV'){
  Get text of body on page
  For (every word of text){
   strip punctuation of word
   check word is not a stop word or nothing
   concatenate word and tV, call this WW

   if(map does not contain key 'word'){
    Put in helperMap (WW, 0)
    Put in map (word, new ArrayList of type TVWC)
    Get Arraylist value at word key of map and...
    add a new TVWC(tB, 0)
   }

   if(helpermap does not contain key 'WW'){
    Get Arraylist value at word key of map...
    add a new TVWC(tV, 0)
    Put in helpermap(WW, size of arraylist at that word)
```

```
    }

    Increment word count of the tV in the associated TVWC

   }
  }
}
```

## makeIndex runs in time O(nm)

We have two loops:

- One goes through all webpages (n)
- The inner loop goes through all words of webpage (m)

All other operations are done in constant time thanks to the use of 2 hash maps

## search()

```
search(String w){
 wordList = get TVWC List at word
 rankingArray = array of type URL Ranking, size of wordList

 For (every TVWC in wordList){
  rank = wordCount in TVWC * in-degree of tV in TVWC
  tV url = new tV(URL, Rank)
  rankingArray[i] = new URLRanking(url)
  }

 Sort rankingArray
 toRet = new ArrayList of type tV
 iterate through rankingArray and put rankingArray[i].tV into toRet

 return toRet
}
```

## search runs in time O(nlogn)

The largest the arraylist could be when we get the Value and Key ‘word’ is n. This would only happen if the word is on every single page we looked at, but it could happen.

- ➔ In this case, making the new array of this size takes **O(n)**
- ➔ Iterating through each page the word is on, computing a ranking, and storing it into the array also takes time **O(n)**
- ➔ Sorting the array using Arrays.sort take time **O(nlogn)** because this will use TimSort and could sort up to n pages.

➔ A final loop iterates through the sorted array and puts it into an arrayList in the form we must return (Tagged Vertex) this takes **O(n)** as well.

In total we have O(n + n + nlogn + n), which is bounded by **O(nlogn)**

## searchWithAnd()

```
searchWithAnd(w1, w2){

rankingsInitial = HashMap of <String, URLRanking>
rankingsFinal = HashMap of <String, URLRanking>

w1WordList = List of (URLs, Incoming Edge Count, and Word Count) for
URLs that contain w1

w2WordList = List of (URLs, Incoming Edge Count, and Word Count) for
URLs that contain w2

For (URL in w1WordList){
      runningRank = # of time w1 appears in URL
      URLRanking rank = Object that holds running rank, in degree,
and URL
      rank.setRunningRank(runningRank)
      rank.setIndegree(URL.inDegree)

      rankingsInitial.put(URL, rank)
}

For (URL in w2WordList){
      if(rankingInitial.contains(URL)){
            We are safe to add this to rankings final because URL
      contains both w1 & w2
            runningRank =
      rankingsInitial.get(URL).getCurrentRanking() + URL.wordCount()
           rankingsFinal.put
      URLRanking rank = Object that holds running rank, in degree,
and URL
      rank.setRunningRank(runningRank)
      rank.setIndegree(URL.inDegree)

      rankingsFinal.put(URL, rank)
    }

}
```

```
For (URL in rankingsFinal){

      Compute the final ranking = URL.running_rank * URL.Indegree

      Copy TaggedVertex from hashmap to rankedList
}

Sort(rankedList)

return rankedList

}
```

## searchWithAnd runs in time O(nlogn)

breakdown of steps and runtime:

1. Grab the list of URLs that contain w1 & w2 in O(1) time from the hashmap
2. Iterate through the list of URLs that contain w1 in O(n)
    a. Compute the running rank O(1)
    b. Add URL to initial HashMap so we can access later in O(1)
3. Iterate through the list of URLs that contain w2 in O(n)
    a. Compute the final running rank O(1)
    b. If the initial HashMap contains this URL then we can add to the final HashMap because we know both w1 & w2 exist for this URL O(1)
4. Copy elements from HashMap to a final ranked list O(n)
5. Sort the final ranked list in **O(nlogn)**
6. Return sorted ranked list

We must iterate through at most n pages multiple times in this algorithm however this is insignificant compared to sorting the final ranked list which takes O(nlogn). Therefore the runtime searching with and is **O(nlogn).**

## searchWithOr()

```
searchWithOr(w1, w2){

rankingsMap = HashMap of <String, URLRanking>

w1WordList = List of (URLs, Incoming Edge Count, and Word Count) for
URLs that contain w1

w2WordList = List of (URLs, Incoming Edge Count, and Word Count) for
URLs that contain w2

For (URL in w1WordList){
```

```
      runningRank = # of time w1 appears in URL
      URLRanking rank = Object that holds running rank, in degree,
and URL
      rank.setRunningRank(runningRank)
      rank.setIndegree(URL.inDegree)

      rankingsMap.put(URL, rank)
}

For (URL in w2WordList){
      if(rankingMap.contains(URL)){
            W1 & w2 both exist for this URL compute an update running
      rank
            runningRank = rankingsMap.get(URL).getCurrentRanking() +
      URL.wordCount()
           rankingsFinal.put
      URLRanking rank = Object that holds running rank, in degree,
and URL
      rank.setRunningRank(runningRank)
      rank.setIndegree(URL.inDegree)

      rankingsFinal.put(URL, rank)
    }
      else{
            This URL has not been added to the map yet
            runningRank = # of time w2 appears in URL
            URLRanking rank = Object that holds running rank, in
      degree, and URL
            rank.setRunningRank(runningRank)
            rank.setIndegree(URL.inDegree)

            rankingsMap.put(URL, rank)
      }


}

For (URL in rankingsFinal){

      Compute the final ranking = URL.running_rank * URL.Indegree

      Copy TaggedVertex from hashmap to rankedList
}

Sort(rankedList)
```

```
return rankedList

}
```

## searchWithOr runs in time O(nlogn)

breakdown of steps and runtime:

1. Grab the list of URLs that contain w1 & w2 in O(1) time from the hashmap
2. Iterate through the list of URLs that contain w1 in O(n)
    a. Compute the running rank O(1)
    b. Add URL to ranking map so we can access later in O(1)
3. Iterate through the list of URLs that contain w2 in O(n)
    a. If this URL is in the map already, we know it contains both w1 & w2
        i. update the running rank O(1)
    b. Else, the URL is not in the map
        i. Compute a running rank O(1)
        ii. Add to the map O(1)
4. Copy elements from HashMap to a final ranked list O(n)
5. Sort the final ranked list in **O(nlogn)**
6. Return sorted ranked list

We must iterate through at most n pages multiple times in this algorithm however this is insignificant compared to sorting the final ranked list which takes O(nlogn). Therefore the runtime searching with and is **O(nlogn).**

## searchAndNot()

```
searchAndNot(String w1, String w2){
 wordList = get TVWC List at word
 rankingList = Arraylist of type URL Ranking

 For (every TVWC in wordList){
  unique word+URL combination notString = w2 + URL in TVWC
  if (notString is not present in helperMap) continue
  rank = wordCount in TVWC * in-degree of tV in TVWC
  tV url = new tV(URL, Rank)
  add new URLRanking(url) to rankingList
  }

 Sort rankingList
 toRet = new ArrayList of type tV
 iterate through rankingList and put rankingList(i).tV into toRet

 return toRet
}
```

### searchAndNot runs in time O(nlogn)

The largest the arraylist could be when we get the Value and Key ‘word’ is n. This would only happen if the word is on every single page we looked at, but it could happen.

- ➔ Iterating through each page the word is on, computing a ranking, and storing it into the list also takes time **O(n)**
- ➔ Sorting the array using Collections.sort take time **O(nlogn)** because this will use TimSort and could sort up to n pages.
- ➔ A final loop iterates through the sorted list and puts it into an arrayList in the form we must return (Tagged Vertex) this takes **O(n)** as well.

In total we have O(n + nlogn + n), which is bounded by **O(nlogn)**

# Choice of Data Structures Justification:

**Crawl.java:**

The big data structure he is of course our graph. While I don’t think this needs justification, the web can be modeled as a directed graph of pages.

We also use a queue for our graph building. The queue lets us build out a graph with breadth first search. This lets use put all the neighbors in for a vertex, and then explore them all first before going any deeper.

We also used a coupling object named **StringDocument**. This stores the URL with its text body parsed from JSoup. We wanted to connect at once point to see if it is a real website (before adding it to the queue) and then when we take the page from the queue we want to look through the body of it for links to be its neighbors. Instead of connecting twice, we store the body of the link on first connect in this object, and then come back to it when we are ready to add its neighbors.

**Index.java:**

The first data structure used in Index.java is a List of Tagged Vertices. We take this exact thing in our constructor so it makes sense to store it locally in the same thing of our own.

We also use two HashMaps, ‘map’ and ‘helper map’. Let’s start with map. **Map** is a Hashmap with Keys of Strings (a word) and a Value that is an Arraylist of TaggedVertexWordCount. **TaggedVertexWordCount** is a coupling object of a TaggedVertex(with a tag that is indegrees of the vertex) and an integer that keeps track of how many times a word has been seen on that URL. The idea is that when you are given a word to search for, you get the Arraylist of TaggedVertexWordCount associated with the word in the Hashmap, and then in that arraylist you have not just the URL and Indegrees, but the number of times the word has been seen there as well. This makes it so we only have to iterate through one list (the word’s arraylist of

URLs) when given a word and we have all the information needed for a ranking calculation.

Thanks to **helperMap**, we can also increment the count for a word on a website in O(1) time in *map*. helperMap is a HashMap with a Key of type String and value of Integer. The key is the concatenation of the word and the URL. Essentially, we are hashing the unique combination of word and url. The value associated with this is the index of the arraylist that the URL appears at in the value of a given key in *map*. This way we can consult helperMap before incrementing at an index to know exactly which index to increment in the word's arraylist, without having to iterate through.

Inside of each of our search functions, we first store the list (or two lists, potentially) of TaggedVertexWordCount as a result from a .get() from our map. This is how they are stored in the map and we just need to iterate through them so this is fine.

Additionally, there an arraylist of type URLRanking (except for the case of the simple search(), where we use an array because we are privy to the exact size that this list needs to be already) **URLRanking** is a tupling object of TaggedVertex (in this case, the tag value is its final rank), an integer to keep track of a URLs In-Degree, and an integer running rank to keep track of rank as we are computing it. As we go through the list for the given word, we compute the ranking and then store that in a new URLRanking object so that we have both the URL and Rank together. Then we can sort this list based on rank, and return just the TaggedVertex part of URLRanking.

There are a few extra parts to searchWithOr and searchWithAnd. In searchWithOr there is a single HashMap rankings. **rankings** has a key of String (URL) and value of URLRanking. We will use this to store the combinatinations of the URL, final rank, In-degree, and running ranking. We can check if the word was already found while iterating through the second word's list of URLs.

In searchWithAnd we use two hashmaps, rankingsInital and rankingsFinal. These have a key of String (URL) and a value of URLRanking. **rankingsInital** is used to store URL1/URLRanking combinations that was found while looking through the first words list of URLs. Then while going through the second words list of URLs we have O(1) access to if we already saw the URL or not. We can store the ones we saw in both to **rankingsFinal**. Then we have O(1) access time to go through rankingsFinal and compute the final.

Finally, we use an arrayList called toRet. At the end of each method we have a list of URLRankings, but need it to be Tagged Vertices, so we just add that part to **toRet** and then return it. It's only real function is to be returned to satisfy the given API

**TwoWayGraph.java:**

I’m not entirely sure we have to justify these, but we use a lot of different data structures so I thought I would do it anyway.

First are the two Linked HashMaps. We realized that since we are working with Strings as vertices and not numbers, a hashmap with keys of type E and the value is a list of type E would work well for our adjacency list. Because we want to be efficient as possible knowing the incoming edges, we keep an inward adjacency list with this same structure. Note: We used LinkedHashMaps because a linked hash map maintains its insertion order of keys, which is important for using index.

Then we have another LinkedHashMap with a key of E and value of integer. This is used for O(1) access time when we are given a url and want to know which index it is assigned to. We also keep an Arraylist of Type E which does the opposite. Using .get() on this arraylist at an index will get the URL at that index. This is important for the interface methods that have index as a parameter.

The next data structure used is an array list of Tagged Vertices. Keeping this updated as we add things to the list will result in us being able to return vertexDataWithIncomingCounts() in O(1) time, instead of having to iterate through each vertex at that time.

Then due to needing the indices as well as the URL for different use cases (like getNeighbors() and getIncoming()), we keep concurrent outgoing and incoming adjacency lists of indices. This is simply done as an arraylist of arraylists of type Integer. Arraylists give us O(1) access time at an index and the flexibility to expand in size somewhat efficiently. Due to our use of Linked HashMaps we can be confident that the URL and Index line up through the whole time.

Finally, the edgesHash is just to have access to all edges in O(1) time. The strings of vertex.from and vertex.to are concatenated and hashed. The value is null for these since we really just care about whether it exists or not. This lets us have a hasEdge(E to, E from) function that we can use not just in this class, but also in Crawl to be more efficient (not connecting when there is an edge already drawn).