PA1 Report

Benjamin Vogel

Crawl Method

The crawl() method was very much a BFS search method with a bit of parallelism, the pseudocode of the method is as follows:

```
Create empty LinkedHashMap<String, Vertex> discovered
Create empty queue Q;
Create Vertex object seedVertex with the seed URL and initialize its depth to 0
Add seedVertex to Q and discovered
Initialize an atomic integer to 0 and politeness semaphore to 1
Do {
        Empty the queue and start a thread for each vertex returned
        Wait until all the threads are finished
} while (Q is not empty)
```

The pseudocode of the thread is as follows:

```
Check if need to wait for politeness policy
Connect to provided URL
For each link from the URL {
        If the link should not be ignored {
                Create a new Vertex object with the link URL
                If the link is not in discovered {
                        If (currentPages < maxPages && currentDepth <= maxDepth) {
                                Add provided Vertex as an ancestor to new Vertex
                                Add new Vertex to discovered and Q
                                Update provided Vertex to have new Vertex as a child
                                Update provided Vertex in discovered
                        }
                }
                Else {
                        Updated new Vertex to be a child of provided Vertex in discovered
                }
}
```

For data structures, I picked a Linked HashMap as the graph representation because not only is get and put for the map O(1) which speeds up the runtime, the linked part makes sure the vertices are ordered. This way, we keep the efficiency of a HashMap while also keeping the integrity of a typical array for ordering. Q was a queue (implemented with a linked list) because a BFS needs a FIFO queue to work correctly and this was the most straightforward implementation. Vertex is an extension of TaggedVertex<String> but also holds much more data such as an ArrayList for both the ancestors and

the children of the vertex (which is very useful in the VertexMap methods), the depth, and the index of the vertex in the graph. The politeness policy uses an AtomicInteger and a Semaphore which allows for mutual exclusion within the politeness policy. They are used to make sure the program waits no more than 3 seconds for the politeness policy and only lets 50 requests be made before enforcing the policy again.

The runtime of the crawl() method is O(m + n) (or edges + vertices), which is the same for a BFS. The main reasoning why this didn't increase or decrease is because discovered is a HashMap with get() and put() methods taking O(1), as well as the queue object having add() and remove() also being O(1). While multithreading is faster than doing the work sequentially, this is primarily because the parallelized part is connecting to the URL in each thread, which is what I saw as the bottleneck of the method. The rest of the algorithm (the actual BFS part) is synchronized in order to prevent dirty reads and dirty writes. Making the crawl() method multi-threaded was designed only to speed up the delay in connecting, and didn't add or remove any runtime to the actual BFS algorithm of crawl().

makeIndex() Method

The makeIndex method is much more straightforward and the pseudocode is as follows:

For each URL in the graph (made by crawl()) {
Check if need to wait for politeness policy
Connect to provided URL
Continue to next URL if it should be ignored
Create new Scanner object
Create an empty HashMap<String, URLOccurrence> called wordOccurrence
While the Scanner has a next word {
Strip the word's punctuation
Continue to next word if it is a STOP word
If the word is not in the HashMap {
Create a new URLOccurrence and add it to the HashMap using the word as the key
} else {
Update the URLOccurrence of the word in the HashMap to have one more occurrence
}
Add it back to the HashMap if the rank is greater than 0
}
Close the Scanner
For each word in wordOccurrence.keys() {
If the word was not found in the inverted index {
Create a new ArrayList<URLOccurrence>, add the occurrence to the new ArrayList, and update the inverted index to have word and the new ArrayList
} else {
Add the word's URLOccurrence to the ArrayList at invertedIndex[word]
}

```
        }
    }
```

The most important data structure for Index that I used was a HashMap<String, ArrayList<URLOccurrence>> called invertedIndex. This allows for looking up the ArrayList to be O(1) and adding to the list to be (technically not but most of the time) O(1). The other methods will iterate through the ArrayList so I couldn't think of another data structure that would work better than an ArrayList. I also had a HashMap wordOccurrence which would be created for each URL and would store the occurrence of words within that URL. I chose a HashMap because adding to it and updating objects within it would be O(1). Also, by waiting until the very end to add all the words back in to the invertedIndex HashMap, I avoided having to search the invertedIndex for every word in every URL in the graph, which saves a large amount of time (Do O(n) once rather than O(n) every word). The last data structure was URLOccurrence which held the URL the word appeared in and calculated the rank of the URL based on that word (helps in the search methods).

The runtime of makeIndex() is O(n * w) where w is the (average) number of words in each vertex. Because we must go to each URL and then parse through most of the word (no STOP words), the runtime of makeIndex can be very slow if there are many vertices and many words per vertex. Updating each occurrence of the word per URL takes O(1) time and migrating the occurrences from the HashMap of that URL to the invertedIndex takes O(u) (or unique words in that URL) time.

Search() Method

The search() method is much smaller than the previous methods and the pseudocode is as follows:

```
Create an empty ArrayList of type TaggedVertex<String> named rankedList
If the inverted index contains word w {
        Get the ArrayList associated with that word from invertedIndex
        For each url in the ArrayList {
                Create a new RankVertex with the url and the rank of the url
                Add the new RankVertex to rankedList
        }
Sort rankedList based on decreasing Rank value
Return rankedList
```

Because you must iterate through each URL for the word regardless, I chose to have an ArrayList as the value for each word in the invertedIndex. This allowed for fetching that ArrayList to be O(1). The rankedList is just a list to be returned based on the signature of the method, and RankVertex is an implementation of TaggedVertex<String> that holds the URL and the rank of the URL for that specific word.

The runtime of the method is O(n*logn):

Creating empty ArrayLIst rankedList: O(1)

Fetching the ArrayList associated with that word: O(1)

Iterating through each URLOccurrence object in the ArrayList: O(n)

Creating and adding a RankVertex object to rankedList: O(1)

Sorting rankedList: O(n* logn) (though it can theoretically be smaller based on the size of n and the algorithm the method decides to use, but I'm assuming Merge Sort).

searchWithAnd() Method

This method has a little bit more involved with it and the pseudocode is as follows:

Create an empty HashMap<String, Integer> called urlRankW1
rankedListW1 = searchNoSort(w1)
rankedListW2 = searchNoSort(w2)
Create an empty List<TaggedVertex<String>> rankedANDList
For each TaggedVertex<String> vertex in rankedListW1 {
urlRankW1[vertex.getVertexData()] = vertex.getTagValue()
}

For each TaggedVertex<String> vertex in rankedListW2 {
If vertex.getVertexData() is in urlRankW1 {
Create a new RankVertex with URL of vertex and rank of vertex.getTagData() + rank of vertex at urlRankW1[vertex.getVertexData()]
Add new RankVertex to rankedANDList
}
Sort rankedAndList based on decreasing rank value
Return rankedANDList

I chose a HashMap to hold the ranked list of W1 because doing the lookup for each word in W2 would only take O(1) each time rather than O(n) each time.

Runtime is O(n * logn) because searchNoSort(w1) and searchNoSort(w2) is O(n) each, adding all the values of w1 to a HashMap is O(n), iterating through words in w2 is O(n) while inside the loop is O(1), and finally sorting here also take O(n * logn).

searchWithOr() Method

This method leverages the fact that OR is the combination of w1 AND w2 + w1 NOT w2 + w2 NOT w1. The pseudocode is as follows:

List<TaggedVertex<String>> searchORList = searchWithAnd(w1, w2)
Add all values of the list returned from searchAndNot(w1, w2) to searchORList

Add all values of the list returned form searchAndNot(w2, w1) to searchORList
Sort searchORList
Return searchORList

Not much to be said about data structures, this one is straightforward.

The Runtime is also O(n * logn) because searchWithAnd is O(nlogn), searchAndNot is O(nlogn) (see below), adding the results is O(n) (searchAndNot and adding is done twice), and sorting is O(nlogn).

searchAndNot() Method

This method takes inspiration from searchWithAnd(), and the pseudocode is as follows:

```
Create an empty List<TaggedVertex<String>> searchNOTList
searchW1List = searchNOSort(w1)
searchW2List = searchNOSort(w2)
Create an empty HashMap<String, TaggedVertex<String>> notMap
For each vertex in searchW1List {
        Add vertex to notMap
}
For each vertex in searchW2List {
        notMap.remove(vertex.getVertexData());
}

searchNotList = notMap.values()
sort searchNotList
return searchNotList
```

I chose a HashMap for the same reason I did it for searchWithAnd(), because adding and removing values in a HashMap take O(1) time, which increases the efficiency of the algorithm.

Runtime is O(nlogn) because search(w1) and search(w2) is O(n) each, adding all of the values of w1 to a HashMap is O(n), iterating through all the words in w2 is O(n) while inside the loop is O(1) (removing based on key is O(1)), and finally sorting is O(nlogn)

searchNoSort()

This is the exact same as search() but without sorting the list, which makes it O(n). (Efficiency)