COMS 535 PA3

Xiaoyun Fu | Raj Gaurav Ballabh Kumar

18th October, 2018

1) **Datastructure used to implement the Weighted Q:**
- We have used Java's inbuilt Priority Queue as the Datastructuer to implement the Weighted Queue. However, there was one issue with the simple implementation - "If multiple elements are tied for least value, the head is one of those elements – ties are broken arbitrarily."
- (PLEASE NOTE: Since we have used the inbuilt datastructure, there is no method called `extract`. The method for retrieval of elements is `poll` or `remove`. The specifications did not enforce that the names of the methods **should** be the same. So we have taken the liberty of using the same names for the methods.)
- This was not the expected behavior from the Weighted Queue. The Weighted Queue was supposed to return the element that was added first to the queue whenever there was a tie among the weights of the elements.
- We got around this issue by using Java's inbuilt Comparator for our Tuple Class.
- Tuple class stores link (the name of the link), weight (the heuristic weight calculated of the link), and the countNum (Assume all nodes are numbered 1..i..N in the order in which they are added to the Priority Queue. This stores i.)
- We overrode the `equals` method in the Comparator Class with a method of our own where the Comparator first compares the Tuples by their weights and if the weights are equal it then compares them by their countNum values. The one with the lower countNum value is pushed up higher in the tree thus ensuring that the link that was added the earliest is the one that is picked first. - Since it is the same Priority Queue as that implemented by Java, the time complexities are also the same.
- This implementation provides O(log(n)) time for the enqueing and dequeuing methods `offer`, `poll`, `remove` and `add`; linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods `peek`, `element`, and `size`.
- To reduce the time taken for searching through the entire Priority Queue, we use a HashTable<String, Value> that stores the links along with their weights. Instead of directly checking for a Tuple in the PQ, we instead first search in the HashTable and replace the Tuple from the PQ only in the case if the current weight of the Tuple is greater than the weight that is stored in the HashTable. This does not improve the worst case runtime, but does make the code faster.

2) **PseudoCode of your Crawling Algorithm**

```
//Initialize count = 0
//Initialize the Queue and VISITED
Add seedUrl to VISITED and BFSQueue
count++
while(! BFSQueue.isEmpty)
    Extract the next Tuple t from the BFSQueue
    Also remove it from the BFSQAsHash
    Download the page at t.link and split the string from <p>
    if(! isTopicSensitive)
        extractLinks(actualText, seed)
    if (isTopicSensitive)
        extractLinks(actualText, seed, true)

extractLinks(actualText, seed)
    for each link l = /wiki/XXXX in the page
        if(! VISITED.contains(l) && VISITED.size() < MAX )
            Add l to VISITED and BFSQueue and BFSQasHASH
            Add edge from t.link to l
            count++
        if(VISITED.contains(l) && VISITED.size() == MAX )
            Add edge from t.link to l
        if(VISITED.contains(l) && VISITED.size() < MAX )
            Add edge from t.link to l

extractLinks(actualText, seed, true)
    for each link l = /wiki/XXXX in the page
        Compute linkWeight of the link l using getHeuristicDistance
        if linkWeight > BFSQasHash.get(l)
            Remove l from BFSQasHash and BFSQueue
            Add new Tuple(l, linkWeight, count) to BFSQasHash
            and BFSQueue
        if(! VISITED.contains(l) && VISITED.size() < MAX )
            Add l to VISITED and BFSQueue and BFSQasHASH
            Add edge from t.link to l
            count++
        if(VISITED.contains(l) && VISITED.size() == MAX )
            Add edge from t.link to l
        if(VISITED.contains(l) && VISITED.size() < MAX )
            Add edge from t.link to l
```

How your procedure will ensure that the graph formed will have exactly MAX many vertices

Since we are adding links to the BFSQueue only until the size of VISITED is less than MAX number of nodes, we ensure that the graph formed will have exactly max many vertices.