Analysis of Performance

Written by Nialls Chavez

Analysis Report:

Header: Here I will attempt to give a summary of why each function runs in the required time. Most of it will be redundant in its description because most of the implementation used one or two types of structures and so their run time will be very similar

/**** MyWebGraph.java ****/

addEdge:
Data structure used: HashMap of hashMaps, it is common knowledge that HashMaps run in O(1) amortized time and only require O(1) amortized space to store due to their internal structure

addNode:
Data structure used: HashMap, it is common knowledge that HashMaps run in O(1) amortized time and only require due to their internal structure.

containsEdge:
Data structure used: HashMap, HashMaps by default are able to do look-up's in O(1) time and only require O(1) space allocation

containsNode:
Data structure used: HashMap, HashMaps by default are able to do look-up's in O(1) time and only require O(1) space allocation

edgeCount:
Data structure used: Global Int, here instead of using the .size method, I just increment a global integer edgeCounter and when called this method just returns that value making it easily runnable in O(1) time.

getBFSIterator:
Data structure used: hashMap of HashMaps, here this iterator runs in O(1) time due to the internal structure of HashMaps, it is known that HashMaps only take O(1) amortized time to traverse

getDFSIterator:
Data structure used: hashMap of HashMaps, here this iterator runs in O(1) time due to the internal structure of HashMaps, it

is known that HashMaps only take O(1) amortized time to traverse.

getNodeID:
Data structure used HashMap, here we return IDMap.get(node)
it is known that a look-up in a HashMap only takes O(1) time
to do a look up on a given key.

neighborCount:
Data structure used: HashMap, here to get the number of neighbors,
I use: graphMap.get(node).keySet().size(),
all together get uses O(1) time, keySet uses O(1) time, as well as
size. which implies that in total, the neighborCount runs in
amortized O(1) time.

neighborSet:
Data structure used: HashMap, here I implement the keySet function
given by the HashMap which is known to run in O(1) time and take
up O(1) space.

neighborSet:
Data structure used: HashMap, here I implement the keySet function
given by the HashMap which is known to run in O(1) time and take
up O(1) space.

size:
Data structure used: Global Int, here instead of using the .size
method, I just increment a global integer nodeCount and when
called this method just returns that value making it easily
runnable in O(1) time.

HashCode:

/**** MyCrawlState.java ****/

addHref:
Data structures used: HashMap and HashMap of HashMaps, here this
function calls addNode, and addEdge, each of which were shown
to run in amortized O(1) time, making this function run in
also O(1) amortized time.

getGraph:
Data structure used: Object, this method returns a reference to the
underlying webGraph object so that it can be used in other classes
this by default runs in O(1) time and requires 0 additional space
because it is just a pointer.

queueLength:

Data structure used: Global Int, here instead of using the .size method, I just increment a global integer queueLengthCounter and when called this method just returns that value making it easily runnable in O(1) time.

getQueue:

Data structure used: LinkedList, retunes a queue and runs in O(1) time and uses O(1) space by the internal representation of the queue.

SaveYourSelf/LoadYourSelf

each require: O(V+E+Q) time and space on disk, where V,E are the sizes of the node set and edge set, respectively, and Q is the length of the to-do queue.
this is due to the nature of saving objects and loading them in.

Empirical analysis

*this data is taken from the output of a run in WebGraphTest.java
it uses the systems run time to calculate the amount of time in nano seconds required to add different numbers of nodes and edges

Time for 10created nodes is: 46000

Time for 100created nodes is: 304000

Time for 1000created nodes is: 3074000

Time for 10000created nodes is: 65810000

Time for 10created edges is: 121000

Time for 100created edges is: 231000

Time for 1000created edges is: 3123000

Time for 10000created edges is: 37091000