1 Data Stuctures for Q and $visited$

Q was implemented as a `LinkedList<String>` object, because a LinkedList can act as a First-in First-out (FIFO) queue structure. Vertices can be added to the front and removed from the end in constant time.

$visited$ was implemented as a `HashSet<String>` object. Adding vertices and searching for vertices are the only two operations needed. Expected runtime for both operations is $O(h(x) + \text{Average Load})$, where $h(x)$ is the time taken to hash the string. Assuming the hash function is good (it is) the average load will be small. Assuming the string lengths are relatively short and don't grow asymptotically (as is the case for the links), we can say the time to compute $h(x)$ is negligible. Therefore the expected runtime for both Insert and Search are approximately $O(1)$.

In short, LinkedList and HashSet were chosen for Q and $visited$ because they both provide constant time operations for the required operations to implement the algorithm.

2 Analysis of WikiCS.txt

Number of edges:

23963

Number of vertices:

500

Vertex with largest out degree:

/wiki/Computer_Science (499 outgoing edges)

Number of strongly connected components:

9

Size of the largest component:

492

3 Data Structures Used in `GraphProcessor`

The main graph was stored in a single `HashMap<String, HashSet<String>>` object. Each String key is a vertex in the graph, and each key's value `HashSet<String>` is the set of vertices representing outgoing edges from the key vertex.

To compute and store Strongly-Connected Components (SCC), a private class called **SCCHelper** was used. It used the following data structures:

- `HashSet<String> visited`
- `PriorityQueue<VertexTime> finishTimes`
- `HashMap<String, HashSet<String>> reversedGraph`
- `HashSet<String> tempSCC`
- `ArrayList<HashSet<String>> stronglyConnectedComponents`

The `VertexTime` class was just a wrapper class for a String and an int, representing the vertex and time values to sort vertexes in the SCC algorithm by decreasing finish times.

After the SCC algorithm runs, the SCC's are stored in the `stronglyConnectedComponents` variable, and all the other data structures are set to `null` so they can be cleaned up by the garbage collector and release memory they were using.

4 Runtimes for `GraphProcessor` Methods

4.1 outDegree(String v)

Since the graph only needs to be initialized once, each call to `outDegree(String v)` only takes $O(1)$ runtime because it simply returns the size of the HashSet at key v in the graph.

4.2 sameComponent(String u, String v)

Iterates over each `HashSet<String>` in the pre-computed strongly-connected components ArrayList and checks if u and v are in the HashSet. If the ArrayList contains k HashSets, the runtime of `sameComponent(String u, String v)` is $O(k)$.

Since $k \leq n$, (where n is the number of vertices in the graph), The worst-case runtime in terms of graph vertices is $O(n)$.

4.3 componentVertices(String v)

Iterates over each `HashSet<String>` in the pre-computed strongly-connected components ArrayList and checks if v is in the HashSet. The HashSet containing v is converted to an ArrayList, which takes $O(j)$ time for a HashSet of size j. Worst-case runtime happens if v is in the last HashSet checked. The runtime for the method is then $O(k + j)$, where k is the number of HashSet objects in the SCC ArrayList.

Since $k + j \leq n + 1$, (where n is the number of vertices in the graph), The worst-case runtime in terms of graph vertices is $O(n)$.

4.4 largestComponent()

Iterates over each `HashSet<String>` in the pre-computed strongly-connected components ArrayList, looks at the size of each, and returns the max size. If the ArrayList contains k HashSets, the runtime for the method is $O(k)$.

Since $k \leq n$, (where n is the number of vertices in the graph), The worst-case runtime in terms of graph vertices is $O(n)$.

4.5 numComponents()

Returns the size of the pre-computed SCC ArrayList, which can be done in constant time.

4.6 bfsPath(String u, String v)

The algorithm looks roughly as follows:

1. Initialize an ArrayList A
2. Create a BFS-Tree starting at u
3. Set *current* to v (in the BFS-Tree)
4. While parentOf(*current*) is not null, add *current* to A and set *current* to parentOf(*current*).
5. Reverse the list A before returning it, so it goes from u to v
6. Return A

Creating a BFS-Tree takes $O(m + n)$ time, where m is the number of edges in the graph and n is the number of vertices. Going up the BFS-Tree from v to u takes $O(n)$ time, and reversing the list takes $O(n)$ time. Therefore, the `bfsPath()` method runs in $O(m + n)$ time.