Joe Kuczek 4/5/2017
CS 311 PA #2

Report: Wiki Crawler and Strongly Connected Components

- Data Structures used for Q and visited:

  Q: I used a Queue that contains strings. The reason for this is because the algorithm we went over in class uses a queue / FIFO method for a BFS traversal. It holds strings because we want to perform the traversal over our wiki links which are strings.

  Visited: I used an Array List that contains strings. Checking if an element exists in an Array List is like checking if an element exists an array. Java's Array List is just a list backed by an array so most operations have a similar time and space complexity to an array.

The following are for the graph WikiCS.txt crawled on /wiki/Computer_Science with max 500 vertices:

- Number of edges and vertices in the graph:

  - Vertices: 500
  - Edges: 23,955

- Vertex with highest out degree (499): /wiki/Computer_Science

- Number of strongly connected components: 1

- Size of largest component: 500

- Data structures built in GraphProcessor:

  - HashMap to store vertices <String, Vertex>
  - HashMap to store outgoing edges (and reversed) <String, ArrayList<String>>
  - HashMap to store finish times for the SCC DFS and distances for BFS traversal <String, Integer>
  - HashMap to store the strongly connected components <Integer, ArrayList<String>>

  I decided to use the <Integer, ArrayList<String>> tuple to store SCC's. The integer key is the "hash code" of the strongly connected component. Each vertex within the strongly connected component will store its component's hash code in a variable called sccKey.

  This way, using componentVertices we can just hash the vertex to find the component. Using sameComponent, we can just compare the vertex's sccKeys to determine if they are in the same component. Finding outdegree, number of components, and max component size is also trivial if we store a variable for maxSCCListSize and update it as we add SCC's to our component listing.

- GraphProcessor method runtime analysis:

  - **int outDegree(String v):**

    O(1), This method simply hashes the string, finds the corresponding outgoing edge ArrayList, then returns the size of that list.

    Total time = Time to compute hash function for v + time to check for null value + time to find the size

  - **boolean sameComponent(String u, String v):**

    O(1), This method hashes the two strings, finds the corresponding vertex objects from the HashMap, then returns true if those two vertices have equal sccKeys or false otherwise. The sccKey is a property of the Vertex object, we can utilize the key to perform fast operations when it comes to SCC's.

    Total time = to compute hash functions for both strings + time to check for null + time to compare two object hash codes

  - **ArrayList<String> componenVertices(String v):**

    O(1), for this all we need to do is retrieve the vertex from our string based on the HashMap. This takes constant time, then we can hash the vertex's sccKey to find the exact component the vertex belongs too. This all takes constant time to retrieve the list of vertices.

    Total time = Time to compute hash function on v + time to check for null + time to compute hash on sccKey

  - **int largestComponent():**

    O(1), we just need to keep track of the max SCC size as we are loading them from a file. We do this by keeping an int variable for max, and updating it whenever the new SCC size is greater than the max value. This all takes constant time.

    Total time = time to return the max variable

  - **int numComponents():**

    O(1), all we need to do here is return the size of our SCC data structure. This will always take O(1) time since it can be stored within the data structure itself.

    Total time = time to return the size of the data structure

  - **ArrayList<String> bfsPath(String u, String v):**

    O(E Log V), where E is the number of edges and V is the number of vertices in the graph. This is the standard time complexity for Dijkstra's shortest path algorithm using a BFS traversal.

    Total time = Initial setup + time to visit all nodes in the graph + time to compute the shortest path