Dustin Welu, Joseph Trom, Thomas Haddy
Programming Assignment 2
ComS 311
04/20/2018

Report for Programming Assignment 2

Using your crawler program, crawl the wiki pages with /wiki/ComputerScience as seedUrl, 100 as max, and empty list as topics. Write the constructed graph to a file named WikiCS.txt. Using NetworkInfluence, compute the top 10 most influential nodes using each of the three algorithms the graph WikiCS.txt Write a report that describes the following:

- **Data structures used for Q and visited. Your rationale behind the choice of data Structures.**
  Q is a queue because the first in first out mechanic will enable the wiki crawler to check all the links of the first page before moving on to any other pages in a breadth first search fashion. Visited is a hash set because it allows us to check if the link has already been visited in O(1) time.

- **Number of edges and vertices in the graph WikiCS.txt**
  There are 100 vertices and 942 edges.

- **Top 10 most influential nodes computed by mostInfluentialDegree and the influence of those nodes.**
  {
  /wiki/Computer_Science,
  /wiki/Computer,
  /wiki/Timeline_of_computing_hardware_before_1950,
  /wiki/Digital_computer,
  /wiki/History_of_Unix,
  /wiki/History_of_computer_hardware_in_Yugoslavia,
  /wiki/Computing_technology,
  /wiki/History_of_computing_hardware,
  /wiki/Computing,
  /wiki/History_of_the_World_Wide_Web
  }

- **Top 10 most influential nodes computed by mostInfluentialModular and the influence of those nodes.**
  {
  /wiki/Computer_Science,
  /wiki/Computer_graphics_(computer_science),
  /wiki/History_of_computing_hardware,
  /wiki/History_of_Unix,
  /wiki/List_of_pioneers_in_computer_science, /wiki/History_of_the_World_Wide_Web,
  /wiki/History_of_computer_hardware_in_Yugoslavia,
  /wiki/Timeline_of_computing_hardware_before_1950,
  /wiki/Computing,
  /wiki/Computing_technology
  }

- **Top 10 most influential nodes computed by mostInfluentialSubModular and the influence of those nodes.**
  {
  /wiki/Computer_Science,
  /wiki/Computer,
  /wiki/Computation,
  /wiki/Procedure_(computer_science),
  /wiki/Algorithm,
  /wiki/Glossary_of_computer_science,
  /wiki/Practical_disciplines,
  /wiki/Computational_complexity_theory,
  /wiki/Computational_problem,
  /wiki/Computer_graphics_(computer_science)]
  }

- **Pseudocode for the constructor and the public methods from NetworkInfluence**

```
NetworkInfluence( String graphData )
{
        Set global variable String filepath to graphData
        Try to make a Graph from the the text file with “filepath” name
        Catch if the file was not found
}

outDegree( String v )
{
        Return vertex v’s number of edges assuming it exists
}

shortestPath( String u, String v )
{
        Hashset of the visited nodes initialized
        Arraylist of the path nodes initialized
        Queue of next nodes to visit initialized
        Add string u to queue
        while(the queue is not empty)
                String cur = the next element in the queue
                if(cur == v)
                        Path  = path to cur
                        Break from the loop
                Else if(cur has not been visited)
                        Add cur to visited
                        Add each edge to the graph with the updated path
        Return path
}

distance( String u, String v)
{
        If ( String u equals String v )
                Return 0
        Let shortestPath be an ArrayList of type String
        Set shortestPath to the calculation of the shortestPath( From u, To v )
        Return shortestPath’s size
}
```

```
distance( ArrayList<String> s, String v )
{
	Let min be the distance( from s's first vertex, to vertex v )
	For ( i from 1 to s's size)
		Let nw be the distance( from s's i-th vertex, to vertex v )
		If ( nw is less than min )
			Min equals nw

	Return min
}

influence( String u )
{
	Let "total" be a float equal to 0
	Let the HashSet "visited" be an empty HashSet of type String
	Let the Queue "toVisit" be a LinkedList with a SimpleEntry with value type
	String vertex and key type Integer distance
	Add String u to toVisit with distance 0

	While ( toVisit is not empty )
		Let "curNode" be the head of toVisit
		If ( curNode's vertex is not in visited )
			Increase total by 1 divided by (curNode's distance^2)
			Add curNode's vertex to visited
			Let "curList" be be a String iterator of vertex curNode's edges---
				---from the adjList
			Go to curNode's first edge
			While ( curNode has another edge )
				Add the edge to toVisit with a distance of the edge's---
					---distance + 1

	Return total
}

influence( ArrayList<String> s )
{
	Let "total" be a float equal to s's size
	Let the HashSet "visited" be an empty HashSet of type String
```

```
        Let the Queue “toVisit” be a LinkedList with a SimpleEntry with value type
        String vertex and key type Integer distance

        For ( i from 0 to s’s size )
                Add s’s current vertex to toVisit with distance 0

        While ( toVisit is not empty )
                Let “curNode” be the head of toVisit
                If ( curNode’s vertex is not in visited )
                        Increase total by 1 divided by (curNode’s distance^2)
                        Add curNode’s vertex to visited
                        Let “curList” be be a String iterator of vertex curNode’s edges---
                                ---from the adjList
                        Go to curNode’s first edge
                        While ( curNode has another edge )
                                Add the edge to toVisit with a distance of the edge’s---
                                        ---distance + 1

        Return total
}

mostInfluentialDegree( int k )
{
        Let “all_nodes” be an empty String ArrayList
        For ( i from 0 to number of vertices )
                Let “visited” be a boolean set to false
                For ( j from 0 to all_node’s size )
                        If ( adjList’s i-th vertex equals all_node’s j-th element )
                                Visited is true
                If ( visited is false )
                        Add to all_nodes the i-th vertex

        Let “most_influential” be a String array of length k
        Let “influential_val” be a float array of length k
        Let index be an int equal to 0
        For ( i from 0 to all_node’s size )
                Let “a” be a float equal to the outdegree( i-th element of all_nodes )
                If ( index is less than k )
                        Most_influential at position index equals all_node’s i-th element
```

```
                Influential_val at position index equals a
                Increase index by 1
        Else
                Let "min" be the 0-th element of influential_val
                Let "ind" be an int equal to 0
                For ( j from 1 to influential_val's length )
                        If ( influential_val at position j is less than min )
                                Min equals influential_val at position j
                                Ind equals j
                If ( a is greater than min )
                        Influential_val to position ind equals a
                        most_influential at 'pos' ind equals all_nodes at position i

    Let "influence" be an empty String ArrayList
    For ( n from 0 to most_influential's length )
        Add most_influential at position n to influence

    Return influence
}

mostInfluentialModular( int k )
{
    Let "result" be an ArrayList be an empty String ArrayList
    Let "infVal" be an array of floats with a length of the number of vertices
    Let "influence" be an empty HashMap with a String key and Float value

    For ( i from 0 to graph's number of vertices )
        infVal's i-th element equals the influence( of adjList's i-th vertex )
        Put infVal's i-th element value with key adjList's i-th vertex into influence

    Sort infVal
    Let "topK" be an empty float ArrayList
    For ( i from infVal's length-k to infVal's length )
        Add the current most influential vertex to topK
    For ( i from 0 to the number of vertices )
        If ( topK found the vertex corresponding to its influence )
                Add vertex to result
        Break if result's size is equal to k
    Return result
```

```
}

mostInfluentialSubmodular( int k )
{
    Let “result” be an ArrayList be an empty String ArrayList
    Let “all_nodes” be an ArrayList be an empty String ArrayList
    For ( i from 0 to graph’s number of vertices )
        Let “visited” be a boolean equal to false
        For ( j from 0 to all_node’s size )
            If ( graph’s i-th vertex is equal to all_nodes j-th vertex )
                Visited equals true
        If ( we haven’t visited a node )
            Add graph’s current vertex to all_nodes

    Let “S” be an ArrayList be an empty String ArrayList
    Let “V” be an ArrayList be an empty String ArrayList
    Let “U” be an ArrayList be an empty String ArrayList

    For ( i from 0 to k )
        For ( n from 0 to all_node’s size )
            Let “in_S” be a boolean equal to false
            For ( y from 0 to S’s size )
                If ( S’s y-th vertex equals all_node’s n-th vertex )
                    in_S equals true

            If ( in_S equals false )
                Clear out V
                Add all of S to V
                Add all_node’s n-th vertex to V

                Let “lessthan” be a boolean equal to false
                For ( m from 0 to all_node’s size )
                    Clear U
                    in_S is equal to false
                        For ( z from 0 to S’s size )
                            If (S’s z-th vertex equals all_node’s--
                              --m-th vertex )
                                in_S equals true
                        If ( n is equal to m and in_S is false )
```

```
                                    Add all of S to U
                                    Add all_node's m-th vertex to U
                                    If ( influence( of U ) is greater than--
                                        --the influence( of V )
                                            Lessthan equals true

                        If ( lessthan is false )
                            Add all_node's n-th vertex to S
                            Break out

            Add all of S to result
            Return result

}
```

- **Analyze and report the asymptotic run time for each of the public methods and the constructor from the class NetworkInfluence. Note that the run times depend on the choice of your data structures. Your grade partly depends on your the asymptotic run time of these methods (which in turn depends on choice of data structures).**

  NetworkInfluence(String graphData) Runtime Analysis:

  Data Structures: A graph in this case is an Adjacency List with V vertices and E edges. The first element in each LinkedList is the vertex and the elements following it are the edges from that vertex. There is also a HashMap containing the vertices' values and their positions in the Adjacency List which allows for O(1) search time for locating a vertex. Constructing an Adjacency List would take O(V + E) time because it will add each vertex into the Adjacency List and also the edges connected to it.

  Runtime:
  $1 + O(V + E) \in O(V + E)$

  outDegree(String v) Runtime Analysis:

  Assuming the vertex exists in the Adjacency List, finding it takes O(1) time because the HashMap has O(1) search time. It returns the vertex's LinkedList(Edges) - 1 which is the value of the vertex's out degree.

  Runtime: $1 \in O(1)$

  shortestPath(String u, String v) Runtime Analysis:

  Assuming that there is a path from vertex u to v, the shortest path would have to do a

BFS

  on the graph which takes O (V + E) time to visit each vertex. In the worst case, the shortest path would contain all of the vertices and edges.

  Runtime: $1 + 1 + 1 + 1 + 1 + O(V + E) \in O(V + E)$

  distance(String u, String v) Runtime Analysis:

  Assuming that there is a path from vertex u to v, the distance is simply the number of nodes between vertex u to vertex v calculated from the shortest path of u and v.

Runtime: $1 + 1 + 1 + O(V + E) \in O(V + E)$

distance(ArrayList<String> s, String v) Runtime Analysis:

Assuming that there is a path from the set of vertices to v, the distance is the minimum shortest path from some vertex $u \in s$ to v. It will have a set of vertices with size S, and v is the one vertex not included in s.
Runtime of distance(String u, String v) is proven to be $O(V + E)$

Runtime: $S * O(\text{distance}) \in O(S * (V + E))$

influence(String u) Runtime Analysis:

This method will take in a string and perform BFS on the graph,
add 1/ (2 ^ distance) to the total, and return the total. So, in the worst case it will go through every vertex in the graph once and performing constant time functions.
The runtime of BFS is proven to be $O(V + E)$

Runtime: $C + O(BFS) * C \in O(V + E)$

influence(ArrayList<String> s) Runtime Analysis:

This method will take in an ArrayList of strings and perform BFS on the graph,
add 1 / (2 ^ distance) to the float value total, and return the total. So, in the worst case it will go through every vertice in the graph once, assuming the entire graph is connected, and perform constant time functions.
The runtime of BFS is proven to be $O(V + E)$

Runtime: $C + O(BFS) * C \in O(V + E)$

mostInfluentialDegree(int k) Runtime Analysis:

This method will loop through every vertex to create a list of all the vertices. It will then loop through that list to find the outdegree of every vertice. The initial k vertices will be placed in an array, then the remaining vertices will cycle through that list to determine if their outdegree is larger than the min in the list. The method will finish by cycling through the array to add every vertice to an arrayList.

The first V will be through cycling through the initial vertices to create a list of all the vertices. The V*k will result from cycling through the vertice list, then cycling through the array to determine if the vertex belongs in the array. The final k will result in adding every element from the array to the arrayList.

Runtime: $V + V*k + k \in O(V*k)$

mostInfluentialModular(int k) Runtime Analysis:

The first part of mostInfluentialModular will loop through all vertices and add each vertex and influence to infVal and influ. Then, it will sort infVal and add the largest k influences to an ArrayList. Finally, it will add the k largest results to the ArrayList result. O(influence) runtime is O(V+E)

Runtime: $V * O(\text{influence}) + V\log(V) + k \in O(V * (V+E) + k)$

mostInfluentialSubmodular(int k) Runtime Analysis:

The method mostInfluentialSubmodular will loop through all vertices and add each vertex to a ArrayList. The method will then proceed into a loop from 0 to k. In that loop, every vertex will then be looped through. Each vertex will compare its influence in union with the resulting set S to the influence of every other vertex in union with S. Every vertex that that is either greater than or equal to the influence of every other vertice will be added to S, up to a cap of k vertices.

The V will be from adding each vertex to the master vertex list. The k will be from cycling through the master loop k times. The V+E * V+E will be from going through the vertex list, the V + V will be from checking the influence of the current vertices to determine the largest $S \cup v$ influence.

Runtime: $V + k * V * V * ((V + E) + (V + E)) \in O(k * V^2 * (V+E))$