Coms 311 Pa2
Report

Data structures used for Q and visited. Our rationale behind the choice of data structures:

We used a Linked List, which implements a Queue, as our Q. The only operations we needed were add() and remove(), which a linked list does in constant time. It is not possible to have a lower asymptotic runtime for those operations.

We used a HashMap as visited. It stores integers (hashed Strings) as the key and Strings (representing the relative url) as the value. We knew we'd be frequently looking up the values and keys that were contained in visited. HashMap typically has a constant lookup time, although the asymptotic lookup time is O(n). The same is true for adding elements. It is therefore faster than any alternative dynamic data structure. It needed to be dynamic because there is no way to no even approximately how many elements it will need to include.

Number of edges and vertices in the graph WikiCS.txt:

Number of Edges: 953
Number of Vertices: 100

Top 10 most influential nodes computed by mostInfluentialDegree and the influence of those nodes:

Nodes: {
/wiki/Computer_Science;
/wiki/Timeline_of_computing_hardware_before_1950;
/wiki/History_of_computing_hardware;
/wiki/History_of_Unix;
/wiki/History_of_the_World_Wide_Web;
/wiki/History_of_computer_hardware_in_Yugoslavia;
/wiki/Computing_technology;
/wiki/Digital_computer;
/wiki/Computer;
/wiki/Computing;
}

Top 10 most influential nodes computed by mostInfluentialModular and the influence of those nodes:

Nodes: {
/wiki/Computer_Science;
/wiki/Timeline_of_computing_hardware_before_1950;
/wiki/History_of_computing_hardware;
/wiki/History_of_Unix;
/wiki/Computing_technology;
/wiki/History_of_the_World_Wide_Web;
/wiki/Computing;
/wiki/History_of_computer_hardware_in_Yugoslavia;
/wiki/List_of_pioneers_in_computer_science;
/wiki/Computer_graphics_(computer_science);
}

Top 10 most influential nodes computed by mostInfluentialSubModular and the influence of those nodes:

Nodes: {
/wiki/Computer_Science;
/wiki/Practical_disciplines;
/wiki/Complex_systems;
/wiki/Purdue_University;
/wiki/Ubiquitous;
/wiki/Human_Genome_Project;
/wiki/Algorithmic_trading;
/wiki/History_of_laptops;
/wiki/Computational_problem;
/wiki/Bletchley_Park;
}

Pseudo code

Constructor:

Create an array list of strings called lines of all lines in the file
set numberOfNodes equal to firstString where firstString is the first string in lines
Create a hashtable that maps Strings to Nodes
For every other string called line in lines:
    Create an array called words of the words in each line using the split function
    If word[0] exists in the hashtable:
        If word[1] exists in the hashtable:
            Add the node returned by providing word[1] as a argument to hashtable to the node returned by providing word[0] as an argument to hashtable.
        Else:
            create a new node using word[1] and add it to the node returned by providing word[0] as an argument to hashtable.
    Else:
        Create a node from word[0]
        Add it to the hashtable

        If word[1] exists in the hashtable:
            Add the node returned by providing word[1] as a argument to hashtable to the node returned by providing word[0] as an argument to hashtable
        Else:b
            create a new node using word[1] and add it to the node returned by providing word[0] as an argument to hashtable.

    Create an edge object using the node returned by providing word[0] as an argument to hashtable and the object returned by providing word[1] as an argument to hashtable.

Outdegree:

Get the node returned by giving the argument given to outdegree as an argument to hashtable
Return the value of that node

shortest Path (String u String v):

Create a stack called stack
Create a queue called Q
Set source equal to the node returned providing u a argument to hashtable.
Set destination equal to the node returned providing v a argument to hashtable.
Unmark all nodes and edges
Set destinationFound to false.

```
For every edge e in source:
        Mark the source  and destination nodes of the edge
        Set depth of e to 1
        Add e to Q
        Push e to Stack
        If destination node of e equals destination:
                Set destinationFound to true
                Break

Create an edge called current
While Q is not empty and destinationFound is false:
        Set current to the next Edge in Q
        For every edge edg in the current's destination's list of edges
                If edg's destination is not marked:
                        Mark edg's destination
                        Add edg to Q
                        Push edj to the stack
                        Set depth of edg to the depth of current + 1
                        If edg's destination equals destination:
                                Set destinationFound to true
                                Break
Set stringPath equal to a new array of strings.
Add v to stringPath
Pop the stack and set current to the result
Add the value of the source of current to stringPath
Create and Edge called temp

While the source of current does not equal source:
         Set temp equal to the result of popping stack
        If current's source equals temp's destination:
                Set current equal to temp
                Add the value of current's source to stringPath
Reverse string Path
Return stringPath
```

Distance (String U, String V):

```
If u equals v:
        Return v
Return the number of items in the list returned by shortest Path ( u , v) -1
```

Distance (List of strings S, string V):

```
Create a list of ints called dists
For each string str in s:
        If shortestPath(str, v) does not return an empty list:
                Add the result of distance(str,v) to dists
If dists is empty:
        Return 0
Return the minimum value in dists
```

Influence(String u):

```
Clear all marked nodes from previous BFS
Clear all stored Influence from previous calculations

Node start = Hashtable.get(u)
Queue Q = new Linked Listed()
start.influence = 1
start.mark
Q.add(start)

While(Q is not empty) {
        Node current = Q.remove()
        For each node with an edge from current {
                If (node is not marked) {
                        node.mark()
                        node.influence = current.influence / 2
                        Q.add(node)
                }
        }
}

Sum the influence of all nodes in the graph
Return the sum
```

Influence (List of strings s):

```
Clear influence for all nodes

Initialize influence of all nodes in s to 1

For all strings string in s {
```

```
        Get Node node from looking up string in the Hashtable
        Initialize Queue Q
        Add node to Q

        While ( Q is not empty ) {
                Set Node current to Q's removed Node

                For all Nodes n with an edge from current {
                        Integer inf =  n.influence
                        If (n.influence < inf) {
                                N.influence = inf
                        }
                }
        }
}
```

Sum influence of all nodes in the graph
Return the sum

mostInfluentialDegree(integer k):

Initialize the list of strings

If the size of the node list is 0, return the empty list

Initialize NodeDegreeComparator
Sort the node list with NodeDegreeComparator

If k > size of the node list, only get node.size results

```
For i = 0 to k / nodes.size {
        Add the node at index length - i - 1 of nodes to the results list
}
```

Return result list

mostInfluentialModular(integer k):

Initialize the list of strings

```
For each node in nodes {
```

```
        Set node.influenceOn = influence(node)
}

Initialize NodeInfluenceComparator
Sort nodes using NodeInfluenceComparator

If k > size of the node list, only get node.size results

For For i = 0 to k / nodes.size {
        Add the node at index length - i - 1 of nodes to the results list
}

Return results list
```

mostInfluentialSubModular (int k)

```
Initialize list strings of type strings
Set notS to a deep copy of nodes

Set float max to 0

While (strings.size < k) {
        For i to notS.size {
                Add node to strings
                Set temp to influence (strings)
                If (temp > max) {
                        maxNode = notS index i
                        max = temp
                        maxIndex = i
                }
                Remove the last index of strings (the one just added)
        }
        Add maxNode to strings
        Remove notS index maxIndex
}

Return strings
```

Asymptotic Runtimes

Constructor:

O(c) where c is the number of characters in the input file
The number of characters in the file will always be greater than the number of edges and could be arbitrarily large.

However assuming that each line is a fixed length the worst case runtime becomes
O(E) where E is equal to the number of edges.

Therefore the runtime = O(E)

outDegree (String v):

This will run in constant time
Therefore the runtime = O(1)

shortestPath (String u, String v):

A BFS is done to until the node with the value of string V is found
The worst case runtime for a BFS is O(V+E) where V is the number of nodes and E is the number of edges

This method also adds edges who's end nodes have not yet been marked to the a stack
At most V edges will be added
The every item is popped off the stack in O(V) time
A list is created and returned in O(V) time with the final path

Therefore the runtime = O(V+E)

Distance (String u, String v):

This method makes a call to ShortestPath which takes O(V+E) time and finds the size of the returned list and subtracts 1 in constant time.

Total runtime = O(V+E)

Distance (List of Strings s, String v):

$$\sum_{k=1}^{S} O(V+e)$$

Where S is the number of elements in S
For every string s the function Distance is called in O(V+E) time
The minimum result is found in O(S)
Total runtime = O(SV+SE)

Influence (String u):

Worst case, the while loop iterates for every single node in the graph, let's call that V.
For each iteration it does the following for loop, where C is a constant and n is the size of the list of nodes reachable by an edge from the current node:

$$\sum_{i=1}^{n} C = C\,n$$

After the while loop, the for loop runs:

$$\sum_{i=1}^{V} C = C\,V$$

V*n*C + V*C = V*n*C
Worst case, n = V.
Total runtime = V*n*C = V*V*C = $O(v^2)$

Influence (List of Strings s):

Let S be the number of nodes corresponding to the strings in s.
Let V be the total number of nodes in the graph.
Let N be the number of nodes reachable by an edge from the current node in the while loop.

The first for loop runs S times performing constant operations:

$$\sum_{i=1}^{S} C = C\,S$$

The while loop runs, at most, V times. The inner for loop runs N times for constant operations.
All of this runs

Inner For loop: $\sum_{i=1}^{N} C = C\,N$

While loop: $\sum_{i=1}^{V} C\,N = C\,N\,V$

Outer for loop: $\sum_{i=1}^{S} C\,N\,V = C\,N\,S\,V$

The last loop: $\sum_{i=1}^{V} C = C\,V$

C*S + S*C*V*N + C*V = C*V*N*S
Worst case, S = V and N = V.
Total runtime = C*V*N*S = V*V*V*C = $O(V^3)$

mostInfluentialDegree (int k):

Java's implementation of sort runs in n * log(n).
The list of nodes is sorted in V * log(V) time, where V is the number of nodes in the graph.

The for loop runs, where k is the given number of nodes to find:

$\sum_{i=1}^{k} C = C\,k$

Worst case, k is equal to V.

Total runtime = V * log(V) + V*C = V*log(V) = O(V * log(V))

mostInfluentialModular (int k):

Let V be the number of nodes in the graph.

The first for loop runs: $\sum_{i=1}^{V} C = C\,V$

The list of nodes of size V is sorted in V * log(V).

The last for loop runs, where k is the given number of nodes to find:

$$\sum_{i=1}^{k} C = C\,k$$

Worst case, k is equal to V.

Total runtime = CV + V*log(V) + CV = V*log(V) = O(V*log(V))

mostInfluentialSubModular (int k):

The first for loop runs V times, where V is the number of nodes in the graph. Link Lists add items in constant time:

$$\sum_{i=1}^{V} C = C\,V$$

The while loop runs as long as the size of the list strings is less than the given integer k. K can be, at most, V. The size of strings starts at 0, increments by one each iteration, and cannot go down. Therefore, the while loop runs, worst case, V times.
The inner for loop runs S times, where S is the size of notS. notS has one element removed at the end of each while loop iteration. At most, it is V.
Inside the inner loop, adding to a Linked List, removing the last element of a Linked List, and multiple assignments are all done in constant operations.

Inner for loop: $\sum_{i=1}^{V} C = C\,V$

While loop: $\sum_{i=1}^{V} C\,V = C\,V^2$

Total Runtime = $CV + CV^2 = CV^2 = O(V^2)$