Geir Matti Järvelä (geirmj11@ru.is)
Þorgeir Auðunn Karlsson (thorgeirk11@ru.is)

Programming Assignment 4: Properties of Social Networks

For the assignment we created three classes, explained further below. All of the classes implement an intelligent lazy solution to each problem. In effect this means that all calculations are done at most once, and only when they are needed. Each method checks first if a solution exists, and if so simply returns a previously calculated solution. This method costs a little bit more in memory, since previous solutions need to be kept, but makes up for it with constant calculating times when duplicating previous calculations, or when two functions use the same results.

Part I - The Centrality class

Degree: The degree() method counts the number of edges to a given Vertex, and returns that number.
Popularity: The popularVertex() method calculates the degree for all vertices and keeps track of the highest degree, returning the corresponding vertex.
Eccentricity: The ecc() method creates a BreathFirstPath for the given node, and then finds the highest distance to every other node, and returns the highest distance.
Center: The center() method goes through the array of eccentrities, calls ecc() for every node that has not been calculated before, and finds the lowest value.
Closeness: The closeness() method creates a BreathFirstPath for the given node, goes through all the paths in the resulting collection and adds their distance to the hollywood variable. It then divides one with that variable, and returns the resulting value.
Closest: The closest() method calculates the closeness of all vertices, keeping track of the highest value, and returns the vertex with that value.
Effective Eccentricity: The effEcc() method generates an arrayList and creates a breathFirstPath for the given node, it then adds all distances from the BreadthFirstPath to the list, sorts the resulting list, and returns the value in the 90% position in the sorted list.
Effective Center: The effCenter() method goes through the list of effective eccentricities and generates the missing values, and returns the node with the lowest value.

The numbers in the table on the right show the complexity of each method, for the first time each method is called. However some of the methods call other methods for every point, for example the closest() method calls the closeness() function for every point in the graph, and since the results are stored, they can be accessed directly thereafter.

Method	Initial Complexity	Subsequent
Constructor	10	constant
Degree	$\sim V$	constant
Eccentricity	$\sim 2V+E$	constant
Effective Eccentricity	$\sim V(\log V)$	constant
Closeness	$\sim 2V+E$	constant
Most Popular Vertex	$\sim V^2$	constant
Center	$\sim V$	constant
Effective Center	$\sim V(V(\log V))$	constant
Closest Vertex	$\sim V(V+E)$	constant

Complexity table: V: Number of Vertices
E: Number of Edges.

Part II - The SymbolCentrality class

The SymbolCentrality class is simply an extention of Centrality. It creates a Symbolgraph class instance, and uses it to create indexed integer keys for each actor.
This allows the class to use the methods of the Centrality class to calculate all necessary values.

Since each function calls the sg.index() method, this adds log(n) complexity to each operation of the Centrality class. In addition more memory is needed because of the SymbolGraph that stores the indices.

Part III - The ExtendedBreadthFirstPaths class

Here we extend the Breath First Search. The generate() method creates a queue and initializes the array numOfPaths[] with a value of -1. Each field represents a single vertex in the graph. The base path

Geir Matti Järvelä (geirmj11@ru.is)
Þorgeir Auðunn Karlsson (thorgeirk11@ru.is)

is initialized to one, since the base node has one path to itself. The base node is then added to the queue, and given a level of zero, in the levels[] array. Each node connected to base is is given a level of one higher, and the number of paths to zero, in order to indicate that they have been added to the queue. We then go through each node on the queue, take the number of paths of each node connected to them that have already been marked, and add that to the number of shortest paths for the current node. We then give all adjacent nodes that have not been marked a level of one higher, and add it to the queue.At the same time we keep track of the vertex with the highest number of paths, and once the method has gone through all vertices, we return that value.
The complexity of this method is $\sim 2V + E$ where V is the number of vertices and E is the number of edges.