# CS6240 Final Report
## Fall 2018

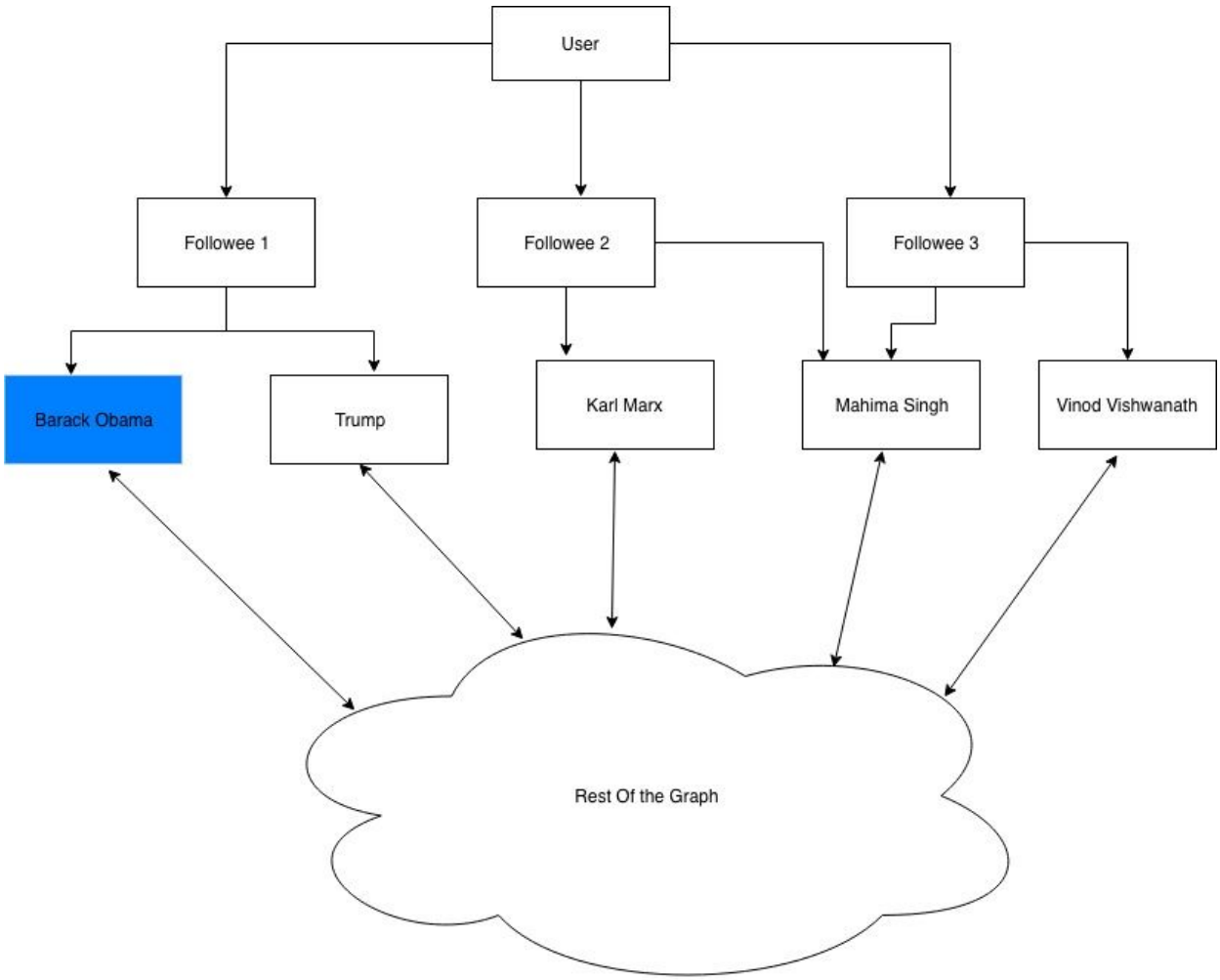
**Team Members** : Vinod Vishwanath, Mahima Singh

---

**Project Overview**

---

Our project goals are two-fold: (1) to find the relative importance of nodes in a large graph using the PageRank algorithm, including elegant handling of sink nodes and spider-traps, and (2) to find the two-hop most influential user followed by every user. This simple recommender system is illustrated in the diagram below:



For the User at the top of the tree, the most influential followee in the 2-hop neighborhood is the user with the highest PageRank value out of ( Barack Obama, Trump, Karl Marx, Mahima Singh, Vinod Vishwanath).

---

## Input Data

---

For this project we used the Twitter graph dataset collected in May 2011. The dataset can be downloaded from the following location:
https://wiki.illinois.edu//wiki/display/forward/Dataset-UDI-TwitterCrawl-Aug2012

The format of the data consists of records as follows:

```
[user_id_1]  [user_id_2]
```

The interpretation of this record is that the user with id `user_id_1` follows the user with id `user_id_2`.

The properties of the dataset are as follows:

| | |
|---|---|
| Nodes | ~ 3 million |
| Edges | ~ 200 million |
| Size | ~ 5.34 GB |

---

## Task Name : Page Rank Over Entire Twitter Dataset

---

Overview : For the first task, we run the PageRank algorithm over entire Twitter dataset, which gives us an idea of the influence of individual nodes. Our goals additionally are to handle Sink nodes (nodes with no outgoing edge) and Spider Traps (a small set of strongly connected components which trap the energy flowing in the graph since they don't point to any node outside the SCC).
To handle sink nodes, we capture the net PageRank of all sink nodes and distribute it evenly in the graph.
To handle Spider traps, we use teleportation with a damping factor of 0.85, which prevents the PageRank from being trapped in sub-SCCs in the graph.

Pseudo-Code :

```
object pageRank {
      def main(args : Array[String]) {
    val iters = if (args.length > 1) args(1).toInt else 10
    val sparkContext = new SparkContext(sparkConfig)
    val graph = sparkContext.textFile(args(0),1)
    val rightEdges = graph.map( s =>
             s.split("\\s+")(1)
              ).distinct()
    val leftEdges = graph.map( s =>
             s.split("\\s+")(0)).distinct()

    val danglingEdges = rightEdges.subtract(leftEdges).map(x=>
(x.toString,"dummy")).groupByKey()
    var edges = graph.map{ s =>
              val node = s.split("\\s+")
              (node(0),node(1))
              }.distinct().groupByKey()

  edges = edges.union(danglingEdges)
  var ranks = edges.mapValues(v => 1.0)
  val dummyRank=sparkContext.parallelize(Seq(("dummy",0.0)))
  ranks = ranks.union(dummyRank)
  val globalRanks = ranks
  edges.cache()
  globalRanks.persist()
    for (i <- 1 to iters) {
      val contribs = edges.join(ranks).values.flatMap{ case (urls, rank) =>
        val size = urls.size
        urls.map(url => (url, rank / size))
      }
      ranks = contribs.reduceByKey(_ + _).mapValues((0.15 + 0.85 * _))
      var noIncoming = globalRanks.subtractByKey(ranks).map(x => (x._1,0.0))

      ranks = ranks.union(noIncoming)
      var delta = ranks.lookup("dummy")(0)
      var size = ranks.count() - 1
    ranks = ranks.map(vertex => if(vertex._1 !="dummy") (vertex._1, (vertex._2
+( delta / size.toDouble))) else (vertex._1,vertex._2))}
ranks.saveAsTextFile(args(2))
```

Algorithm and Analysis :

Well since it was such a heavier dataset and there is a join operation on graph at every iteration, it was prudent to cache graph RDD, which helps to optimize the performance. Also in every

iteration, there is a constant subtract operation on global page rank RDD to keep track of nodes with no outgoing links.

Experiments

AWS Cluster Sizes

- 1 Master 5 Worker nodes
- 1 Master 2 worker nodes

Machine Configuration : m5.xlarge
Number of Iterations : 5

SpeedUp

| Cluster Size | Execution Time |
|---|---|
| 5 workers 1 Master | 24 minutes |
| 2 workers 1 Master | 76 minutes |

The results show almost ideal speedup when executed on 10 workers as compared to 5.

ScaleUp

| File Size | Execution Time |
|---|---|
| 1 GB - 5 workers 1 Master Cluster | 5 minutes |
| 5 GB - 5 workers 1 Master Cluster | 24 minutes |

Result Sample

```
[user-id] [page-rank]
```

All output runs are available at :
https://drive.google.com/drive/folders/1wAIxc2fwLRV337DjHZYzL5UR83NGBHAQ?usp=sharing

## Task Name : Recommendation of most influential users in n-hop neighborhood

Overview : The goal of this task is to do a 2-hop-neighborhood search for each user to find the most influential user within two hops. To successfully complete the task, we use a clever approach: we find the reverse of the graph, where the direction of every edge is inverted, and in a MapReduce job, we pass the PageRank of every node to each of its children in the Map phase. In the Reduce phase, each node compares the values received and chooses the top neighboor. Two iterations of this task produces a 2-hop neighborhood search.

Therefore, this task consists of three sub-steps:
(i) Reverse the edges of the graph
(ii) Perform an equi-join on the now left side of the edge with the page-rank dataset
(iii) Perform n-hop-neighborhood search

Pseudo-Code :

- 2.1 Reversing the graph

```
map(from, to) {
      emit(to, from)
}
Reduce ( Node node, Iterable<Node> nodes) {
      adjacent_nodes = collect_into_array(nodes)
      emit(node, adjacent_nodes)
}
```

- 2.2 Joining graph and pagerank dataset

```
Map(Text value) {
      if(value is from Page rank) {
            node_id, page_rank = parse(value)
            emit(node_id, (dummy, page_rank))
      }
      else {      // value from graph
            Graph node = new GraphNode()
            node.node_id = value.nodeId;
            node.adjacencyList = value.getAdjacencyList;
            emit(value.nodeId, node)
      }

Reduce(nodeId, Iterable<GraphNode> values){
      node = Null
      page_rank = -1
      Iterate through values
            if(value.nodeId == -1) //means page rank
                  page_rank = value.pagerank
```

```
            else // graph structure
                  node = value
      node.page_rank = page_rank // RS-Join
      emit(node)
```

- 2.3 Breadth first neighborhood search for Influential users

```
Map(node):
      emit(node_id, node) // Preserve graph structure

      dummy_node = new GraphNode(node_id=-1)
      dummy_node.top_neighbor = top_neighbor(node)
      for child_node_id in node.adjacency_list:
            emit(child_node_id, dummy_node)

Reduce(node):
      neighbors = Empty Array
      node = null

      for value in values:
            if value is node:
                  node = value
            else
                  neighbors.add(value)

      node.max_neighbor = max(neighbors)
      emit(node)
```

Algorithm and Analysis:
At first glance, an n-hop-neighborhood search for every node in the graph looks like a complex problem. A basic implementation in a non-parallelized program might make use of a depth-limited BFS search for every node.
Using MapReduce and using the graph structure elegantly by reversing the direction of edges, we are able to perform this task much more efficiently. The complexity of the MapReduce program thus becomes the same as a single node BFS. The size of the intermediate results can be very high, especially in graphs with a high branching factor, but these are limited in the reduce phase of each iteration by choosing the top neighbor and discarding other results.

Experiments

AWS Cluster Sizes

- 1 Master 15 Worker nodes
- 1 Master 10 worker nodes

Machine Configuration : m5.xlarge

SpeedUp

| Cluster Size | Execution Time |
|---|---|
| 10 worker nodes 1 master | 46 minutes |
| 15 worker nodes 1 master | 31 minutes |

Result Sample

All output runs are available at :
https://drive.google.com/drive/folders/1wAIxc2fwLRV337DjHZYzL5UR83NGBHAQ?usp=sharing

## Conclusion

Through this project, we show an elegant way to handle problems associated with finding the PageRank of nodes in a graph, such as Sink Nodes and Spider Traps. Secondly, we show a clever way to exploit the structure of graphs to efficiently perform N-hop-neighborhood search for every node in a graph.
Such a program could be used by a system like Twitter to find follow-recommendations to its users, since the 2-hop boundary consists of users that user X does not follow directly.
This approach could further be enhanced to find the top-k neighbors, to provide multiple recommendations.
Another enhancement would be to increase the number of hops, which simply corresponds to one additional iteration of the Neighborhood Search task.

## Citation

The source code has been cited in the submitted code which can be found at:
https://github.ccs.neu.edu/cs6240f18/2Spark----2Spark