Sādhanā Vol. 40, Part 6, September 2015, pp. 1821–1843. © Indian Academy of Sciences

Parallelization of game theoretic centrality algorithms

M VISHNU SANKAR* and BALARAMAN RAVINDRAN

Department of Computer Science and Engineering, Indian Institute of Technology, Madras, Chennai 600 036, India
e-mail: vishnusankar1512@gmail.com; ravi@cse.iitm.ac.in

MS received 18 September 2014; revised 9 July 2015; accepted 19 July 2015

Abstract. Communication has become a lot easier with the advent of easy and cheap means of reaching people across the globe. This has allowed the development of large networked communities and, with the technology available to track them, has opened up the study of social networks at unprecedented scales. This has necessitated the scaling up of various network analysis algorithms that have been proposed earlier in the literature. While some algorithms can be readily adapted to large networks, in many cases the adaptation is not trivial. In this work, we explore the scaling up of a class of node centrality algorithms based on cooperative game theory. These were proposed earlier as an efficient alternatives to traditional measure of information diffusion centrality. We present here distributed versions of these algorithms in a Map-Reduce framework, currently the most popular distributed computing paradigm. We empirically demonstrate the scaling behavior of our algorithm on very large synthetic networks thereby establishing the utility of these methods in settings such as online social networks.

Keywords. Hadooop; centrality; social networks; parallelization; game theory; map-reduce.

1. Introduction

Interactions between people in social media, telephone calls between phone users, collaboration between scientists and several other social interactions have significant correlation with the behavior of people. Decisions taken by people are influenced by their social interactions. A network is a natural way of modeling these social interactions. The social actors are modeled as nodes and the dyadic relationships between them are modeled as edges. Analysis on these *social networks* have helped us in understanding several social phenomena (Carrington *et al* 2005) such as information spread/epidemic spread through the network (Lerman & Ghosh 2010), stability of the network and formation of new links (Linyuan & Tao 2011). It has also improved personalized recommendations and personalized web searches.

*For correspondence

Finding important nodes/edges in the network is one of the chief challenges addressed by social network analysis. The measure of importance of nodes/edges, known as centrality (Bonacich 1987), varies depending on the context. Generally, a centrality measure which is optimal in one of the context will be sub-optimal in a different context. The popular centrality measures are degree centrality, closeness centrality, betweenness centrality and page rank centrality. Degree centrality of a node is the number of links incident upon that node. It is useful in the context of finding the single node which gets affected by the diffusion of any information in the network. It follows from the fact that the node with high degree centrality has the chance of getting affected from many number of sources. Closeness centrality of a node is the sum of the inverse of the shortest distance from the node to every other node in the network. In applications such as package delivery, a node with high closeness centrality can be considered as the central point. Betweenness centrality of an edge is the fraction of shortest paths between all pairs of nodes in the network that passes through this edge. A node with high betweenness centrality will typically be the one which acts as a bridge between many pairs of nodes. Page rank centrality of a node depends on the number and the quality of the neighbors who have links to the node. One of the popular applications of page rank centrality is finding the relevant page from the web.

Information cascade is another important context of measurement of centrality. The behavior of the neighbors influences the behavior of an individual to a larger extent. It is useful in several application such as viral marketing where the initial set of influencers determine the success (Bass 1969; Brown & Reingen 1987; Domingos & Richardson 2001; Richardson & Domingos 2002), identification of critical nodes in a power systems where the failure of a critical node may cause a cascading failure leading to failure of the entire network (Asavathiratham *et al* 2001), in modeling the epidemic spread and in finding the critical nodes in computer networking. The traditional centrality measures do not consider the collective importance of nodes and hence fail to identify optimal nodes in the context of information diffusion. A greedy algorithm was introduced and was proven to model the context of information cascade better than the traditional centrality measures (Kempe *et al* 2003). But, greedy algorithm is of exponential complexity and hence it cannot be applied on large networks. Game theoretic centrality algorithms model the importance of nodes when combined with other nodes in the network (Narayanam & Narahari 2008, 2011). This makes it suitable to be applied in the context of information diffusion. Polynomial time algorithms to compute game theoretic centrality were introduced by Tomasz Michalak *et al* (2013).

Huge volume of data is being generated by online social media, online businesses, scientific research, etc. (McKinsey Global Institute 2011). As of 2013, Netflix was having 3.14 PB of videos and Facebook was having 240 billion photos. The capacity of the world to exchange information was 281 PB in 1986, 471 PB in 1993, 2200 PB in 2000 and 65000 PB in 2007 (Hilbert & Löpez 2011). Further, the availability of easy tracking mechanisms of these interactions has led to the development of huge networks. With the advent of technologies, we are able to store these very large amount of data. The network size have grown by leaps and bounds that even small order polynomial time algorithms need to be parallelized to have a better scaling behavior. In this article, we introduce the techniques to parallelize the game theoretic centrality algorithms.

There are many parallelization frameworks available to parallelize graph algorithms. We chose map-reduce technique to parallelize our algorithms. As of 2013, more than half of the fortune 50 companies use map-reduce. Facebook uses map-reduce cluster to process 100 PB of data which is the largest in the world by 2013. It is also available commercially in cloud under the names Elastic map-reduce and Azure HDInsight offered by Amazon and Microsoft, respectively. Apart from being widely used, it has other advantages such as easy to use, fault tolerance, high scalability, etc.

Rest of the paper is organized as follows. In section 2, we give a brief overview of map-reduce programming model and hadoop. In section 3, we present the challenges in the parallelization using hadoop. In section 4, we present the five game-theoretic centrality algorithms proposed in Tomasz Michalak *et al* (2013) and the techniques to parallelize these algorithms in hadoop. In section 5, we evaluate the performance of our parallelized algorithms using the synthetic and real world data sets. Conclusions are provided in section 6.

2. Map-reduce and Hadoop

Map-reduce (Dean & Ghemawat 2004) is a parallel programming model for data intensive applications. Every map-reduce task starts with a map phase. The input and output to this phase are key-value pairs. One key-value pair is read at a time and one or many key-value pairs are written in the output.

$$(Key1, Value1) \rightarrow (list(Key2, Value2))$$

An optional combiner can be used at the end of the map phase to reduce the size of the output of the map phase. This reduces the amount of data to be transferred over the network which otherwise would be a bottleneck. This phase is followed by the sort and shuffle phase. The output of the map (or combiner) is sorted and are sent to the respective reducers based on a partition algorithm. The default partition algorithm uses the hash value of the key to make sure that same key goes to a single reducer. It can also be overridden by a custom partitioning algorithm. Once the sort and shuffle phase ends, the reduce phase begins. The input to this phase is the key and the list of values corresponding to that key. For each key, one or many key-value pairs are written into the output by the reducer.

$$(Key2, list(Value2)) \rightarrow (list(Key3, Value3))$$

Hadoop (Apache Software Foundation 2011) is an open source implementation of the map-reduce programming model. It follows a master–slave architecture where the master (Job Tracker) takes up a job, assigns a part of the job (task) to each of the slaves (Task Tracker) and tracks the progress of the job from the reports of the slaves to the master. Generally, slaves are the data nodes wherein the input data is stored and processed. But, master can also act as a data node and do computation. The input file is broken into several chunks of equal size (except the last chunk) and are distributed among the data nodes in the Hadoop distributed file system (HDFS)

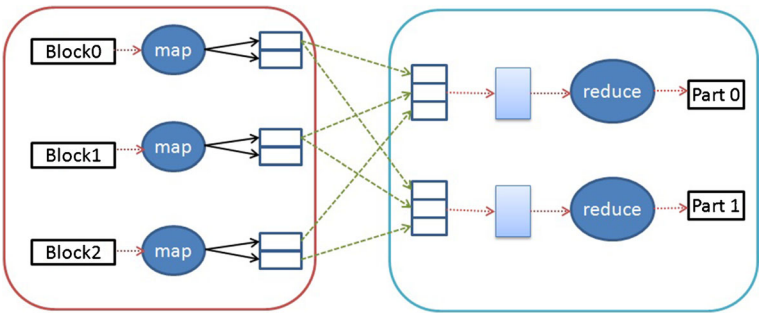

Figure 1. Map-reduce model with 3 mappers and 2 reducers.

(Borthakur 2007). Each block or a chunk is replicated and are stored in many machines providing fault tolerance. Every machine can run any number of map or reduce tasks at a time which is configurable (figure 1). This framework is used to parallelize the five centrality algorithms proposed in Tomasz Michalak *et al* (2013) to run them on big data.

2.1 *Challenges in parallelization*

The challenges in parallelization of graph algorithms included representation, programming perspective and optimization of running time.

Representation: In general, large graphs are represented in the edgelist format. In this format, each line of the input file is an edge. For example, edge AB is represented by "A [space] B" where A and B are the nodes forming the edge.

Input format for the graph represented in figure 2 could be:

```
1 2
2 3
2 4
2 5
3 5
```

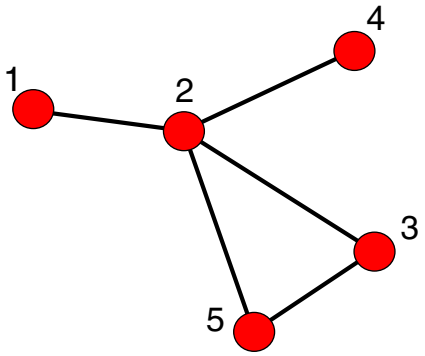

Figure 2. An example network which is unweighted.

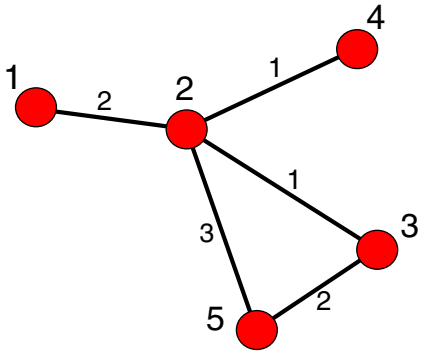

Figure 3. An example network which is weighted.

Input format for the graph represented in figure 3 could be:

```
1 2
1 2
2 3
2 4
2 5
2 5
5 2
3 5
5 3
```

Programming perspective: There are two challenges from the programming perspective.

(i). The input file is partitioned into equal chunks and are distributed across the machines in the cluster. So, the information related to a particular node may be stored partially in many machines. For example, edges "1 2" and "1 3" may be present in two different machines. In this case, we say that the information about node 1 is partially available in each of the two machines.

(ii). The mapper reads the input chunk (or an input file) in a sequential manner such that it reads only one line at a time. In our case, since each line corresponds to an edge, we say that map processes one edge at a time. Essentially, map will have knowledge only about the edge which is getting processed at that time and will not have any information about the other edges in the network. Similarly, the reducer processes one key at a time and will have information about only that particular key and its values at a time.

Optimization of running time: In the map-reduce setting, the optimization for the running time of an algorithm can be done only in two areas. One is to reduce the number of map-reduce phases. As the number of map-reduce phases reduce, amount of computation decreases and hence there will be a decrease in the running time of the algorithm. Another is to reduce the amount of intermediate data i.e., the output of map. This intermediate data have to be shuffled and sorted and have to be sent to the reducer through the network. If the intermediate data is large, it will become a bottleneck and the algorithm slows down.

3. Algorithms

In this section, we describe the five game theoretic centrality algorithms proposed in Tomasz Michalak *et al* (2013) and the techniques to parallelize them. The class of centrality measures proposed in Tomasz Michalak *et al* (2013) are defined using cooperative games on networks. The following definitions describe the general methodology used for characterizing these measures.

Definition: A *cooperative game* is characterized by a set of players N and a payoff function $v : 2^N \to \mathscr{R}$, that assigns a real value to any subset of players, or *coalition*.

Definition: The *Shapley value* of a node in a cooperative game is the expected marginal contribution of the node to any coalition. This is determined by computing the expected increase in the value of a randomly chosen coalition when this node is added to it.

The notion of Shapley value is one of the central concepts of cooperative game theory and is typically used to apportion payoffs to members of a coalition in a stable way. For a more formal definition of Shapley value, see Gibbons (1992). What is of importance in this context is that the Shapley value considers a node's contribution in relation to groups of other nodes and does not look at a node in isolation. This allows us to define newer notions of centrality as explored in Tomasz Michalak *et al* (2013).

In the context of networks, the nodes of the network are considered as agents in a cooperative game that models the process of influence propagation. For different models of propagation, we get different definition of games. In particular (Tomasz Michalak *et al* 2013) introduce the notion of *fringe* of a coalition — the set of nodes influenced by a coalition at least to some extent is known as the fringe of the coalition. By changing the definition of the fringe we can model different influence propagation mechanisms. The Shapley value of a node now is the marginal contribution that a node makes to the fringe across all coalitions that the node is a part of. The Shapely value can then be used as a centrality measure in the context of information diffusion. It has been shown (Tomasz Michalak *et al* 2013; Narayanam & Narahari 2011, 2008) that Shapley value based centrality measures are equally effective if not better than classical notions of centrality for choosing most influential users under appropriate diffusion models. More importantly, these centrality measures are more efficient to compute than classical measures. For more details of the approach, refer to Tomasz Michalak *et al* (2013).

3.1 *An example: Degree centrality*

Degree centrality is the easiest of the centralities in terms of computation. The degree centrality of a node is defined as the degree of the node. Let the input file be in the edgelist format. The degree centrality can be calculated in one map-reduce phase as in algorithm 1.

Algorithm 1. Degree centrality.

Input: Edgelist of $G(V,E)$
Output: Degree Centrality

```
class MAPPER
    method MAP(lineNumber, edge)
        (nodeA,nodeB) = edge.split()
        EMIT(nodeA, 1)
        EMIT(nodeB, 1)

class REDUCER
    method REDUCE(node, list(values))
        degree = 0
        for all value ∈ values do
            degree += value
        end for
        EMIT(node, degree)
```

The input file gets partitioned into several chunks and every machine in the cluster will have few chunks of this file. In the map phase, mapper is started in every machine and each mapper processes one chunk of the input file. Each chunk comprises a portion of the edgelist and all the chunks together form the edgelist of the network. The mapper processes one line at a time i.e., the map function gets called for every input line. In this case, since each line is an edge of the network, we can say that it processes one edge at a time. The map function tokenizes the input line into tokens. The tokens are the two nodes (say A and B) which had formed the edge. For every edge, two (key,value) pairs are written to the output. They are:

A 1
B 1

Once every mapper in all the machines has completed processing their chunk, the output of all the mappers go to a sorting and shuffle phase. In this phase, all the map outputs are sorted and a partitioner sends them to the reducer. The partitioner uses a hash function and makes sure that all the (key,value) pairs which have the same key goes to the same reducer. In reducer, the values corresponding to a single key are put together in a list. The reduce function gets called for every (key, value) pair. It aggregates the values in the list corresponding to each key. Each key here is a node in the network and the aggregated sum is the degree of that node. So, the output of the reducer is just the pair (node, degree). The output file from all the machines are concatenated which gives the degree centrality of all nodes in the network.

3.2 *Game 1: Number of agents at-most 1 hop away*

The first game theoretic centrality algorithm computes centrality by considering the fringe as the set of all nodes that are reached in at-most one hop. Algorithm 2 gives the steps to calculate this centrality. It has a running time of $O(V + E)$.

Algorithm 2. Game 1.

Input: Unweighted graph $G(V, E)$
Output: Centrality values of all nodes

for all $v \in V$ **do**
  Centrality(v) = $\frac{1}{1+\text{degree}(v)}$
  for all $u \in$ Neighbors(v) **do**
    Centrality(v) += $\frac{1}{1+\text{degree}(u)}$
  end for
end for

From algorithm 2, we observe that in order to compute the centrality of a node, we need to find degree of the node and its neighbors. The intuition behind this algorithm is that the centrality of a node will be higher not just when the node has a high degree but also when the node has more neighbors who have low degree. We parallelize this algorithm using two map-reduce phases.

In the first map-reduce phase, we calculate degree of all the nodes and their neighbors in a parallel fashion as described in algorithm 3. The map-phase of this stage do not do any computation

but modifies the input edge in such a way that degree of nodes can be calculated from the edgelist. For every edge in the input file, two lines are written in the output. Let "A B" be the line which is getting processed by the mapper. The first line of the output will have A as the key and B as the value. This indicates that B is a neighbor of A. Since the graph is undirected, A is a neighbor of B. So, the second line of the output will have B as the key and A as the value. The combiner is not needed in this phase since we are not interested in merging the values corresponding to the same key. We use the default partitioner which makes sure that all the (key, value) pairs with the same key goes to the same reducer.

The reducer in this stage receives nodes as keys and their neighbors as values. We calculate the degree of nodes by counting the number of their neighbors. Note that, if a weighted graph is given as input, duplicate representations of the same neighbor will be found. We find the duplicates using the hash value of the neighbors and ignore the duplicates while counting the number of neighbors.

Algorithm 3. First map-reduce phase of game 1.

```
class MAPPER
    method MAP(lineNumber, edge)
        (nodeA, nodeB) = edge.split()
        EMIT(nodeA, nodeB)
        EMIT(nodeB, nodeA)

class REDUCER
    method REDUCE(node, list(neighbors))
        degree = length(neighbors)
        marginalContribution = 1/(1+degree)
        EMIT(node, marginalContribution)
        for all neighbor ∈ neighbors do
            EMIT(neighbor, marginalContribution)
        end for
```

Algorithm 4. Second map-reduce phase of game 1.

```
class MAPPER
    method MAP(lineNumber, value)
        (node, marginalContribution) = value.split()
        EMIT(node, marginalContribution)

class REDUCER
    method REDUCE(node, marginalContributions)
        centrality = 0
        for all marginalContribution ∈ marginalContributions do
            centrality += marginalContribution
        end for
        EMIT(node, centrality)
```

Since, we are interested in the calculation of the marginal contributions of each node, which is $\frac{1}{1+degree}$ in this case, we write the marginal contributions of the nodes in the output instead of

their degrees. Every node needs its marginal contribution in order to calculate its own centrality. So, the reducer writes the node as the key and its marginal contribution as value. Also, all the neighbors need the marginal contribution of the node in order to calculate their centrality. So, the reducer writes every neighbor as key and marginal contribution of the node as value for all the neighbors.

In the second map-reduce phase, centrality values of all the nodes are calculated. The mapper in this phase does not do any computation. It reads the input from the file written by the reducer of first phase and tokenizes them into (key,value) pairs and writes them to the output. The reducer in this phase gets nodes as keys and the marginal contributions of all their neighbors as values. So, the reducer aggregates the values corresponding to the nodes which will give the centrality of the nodes. A combiner is used in this phase which does the same computation as the reducer.

3.3 *Game 2: Number of agents with at least k neighbors in C*

The second game theoretic centrality algorithm computes centrality by considering the fringe as the set of all nodes that are either in the coalition or that are adjacent to at least k nodes which are already in the coalition. Algorithm 5 gives the steps to calculate this centrality. It has a running time of $O(V + E)$.

Algorithm 5. Game 2.

Input: Unweighted graph $G(V, E)$
Output: Centrality values of all nodes

for all $v \in V$ **do**
  k(v) = Random(1,degree(v)+1)
end for
for all $v \in V$ **do**
  Centrality(v) = min $\left(1, \frac{k(v)}{1+\text{degree}(v)}\right)$
  for all $u \in$ Neighbors(v) **do**
    Centrality(v) += max $\left(0, \frac{\text{degree}(u) - k(u) + 1}{\text{degree}(u)(1+\text{degree}(u))}\right)$
  end for
end for

From algorithm 5, we observe that in order to compute the centrality of a node, we need to find the degree of the node and its neighbors. The intuition behind this algorithm is that every node will be influenced only when k of its neighbors are already influenced. This k is the threshold which varies from 1 to degree(node)+1. A threshold of degree(node)+1 for a node indicates that the node cannot be influenced even when all its neighbors are influenced. We parallelize this algorithm using two map-reduce phases.

The first map-reduce phase of this algorithm is given in algorithm 6. The only difference between the first phase of game 1 and game 2 algorithms is the way in which the marginal contributions are calculated. So, the mapper does the same job as algorithm 3. In the reduce

phase, marginal contributions for a node is calculated by the formula $\frac{k}{1+degree}$ and the marginal contribution of its neighbors are calculated by the formula $\frac{degree-k+1}{degree(1+degree)}$.

Algorithm 6. First map-reduce phase of game 2.

```
class MAPPER
    method MAP(lineNumber, edge)
        (nodeA, nodeB) = edge.split()
        EMIT(nodeA, nodeB)
        EMIT(nodeB, nodeA)

class REDUCER
    method REDUCE(node, list(neighbors))
        degree = length(neighbors)
        marginalContribution = k/(1+degree)
        EMIT(node, marginalContribution)
        marginalContribution = (degree-k+1)/(degree(1+degree))
        for all neighbor ∈ neighbors do
            EMIT(neighbor, marginalContribution)
        end for
```

The second map-reduce phase of this algorithm is essentially the same as that of the second phase of game 1 (algorithm 4) which aggregates the marginal contributions to obtain the centrality.

Algorithm 7. Game 3.

Input: Weighted graph $G(V, E)$
Output: Centrality values of all nodes

```
for all v ∈ V do
    extNeighbors(v) = {}
    extDegree(v) = 0
    for all u ∈ V such that u≠v do
        if Distance(u) < d_cutoff then
            extNeighbors(v).push(u)
            extDegree(v)++
        end if
    end for
end for
for all v ∈ V do
    Centrality(v) = 1/(1+extDegeee(v))
    for all u ∈ extNeighbors(v) do
        Centrality(v) += 1/(1+extDegree(u))
    end for
end for
```

3.4 *Game 3: Number of agents at-most d_{cutoff} away*

The third game theoretic centrality algorithm computes centrality by considering the fringe as the set of all nodes that are within the distance of d_{cutoff} from the node. Algorithm 7 gives the steps to calculate this centrality. It has a running time of $O(VE + V^2 log(v))$.

Algorithm 8. First map-reduce phase of game 3.

```
class MAPPER
    method MAP(lineNumber, Edge)
        (nodeA, nodeB) = Edge.split()
        EMIT(nodeA, nodeB)
        EMIT(nodeB, nodeA)

class REDUCER
    method REDUCE(node, list(neighbors))
        for all neighbor ∈ neighbors do
            currentWeight = Map(neighbor→weight).get(neighbor)
            Map(neighbor→weight).delete(neighbor,currentWeight)
            Map(neighbor→weight).add(neighbor,currentWeight+1)
        end for
        for all neighbor ∈ neighbors do
            EMIT(node, neighbor)
            weight = Map(neighbor→weight).get(neighbor)
            Map(weight→neighbors).add(weight, neighbor)
        end for
        weights = keys(Map(weight→neighbors))
        for all weight ∈ weights do
            neighborString = null
            for all neighbor ∈ neighbors do
                neighborWeight = Map(neighbor→weight).get(neighbor)
                newWeight = weight + neighborWeight
                if newWeight < d_cutoff then
                    neighborString.concat(neighbor+":"+newWeight+":")
                end if
            end for
            Map(weight→neighborString).add(weight, neighborString)
        end for
        for all neighbor ∈ neighbors do
            neighborWeight = Map(neighbor→weight).get(neighbor)
            neighborString = Map(weight→neighborString).get(neighborWeight)
            EMIT(neighbor, neighborString)
        end for
```

From algorithm 7, we observe that in order to compute the centrality of a node, we need to find the extDegree and extNeighbors of every node. This algorithm is an extension of game 1 to the weighted networks. The intuition behind this algorithm is that a node can be influenced by an influencer only when the distance between the node and the influencer is not more than d_{cutoff}. This d_{cutoff} is generally fixed to a constant value and we have fixed it as 2 in our parallelization. For an unweighted graph, all the nodes which are one and two hops away will form the extNeighbors but for a weighted graph, it depends on the weights of the edges of the graph. We parallelize this algorithm using four map-reduce phases.

The first map-reduce phase of this algorithm is given in algorithm 8. The map phase of this algorithm does the same job as the map phase of the first map-reduce phase of game 1 and game 2 algorithms. The reducer in this phase gets nodes as keys and list of their neighbors as values. These neighbors are one hop neighbors i.e., they are connected to the node by a single edge. The number of times a neighbor appears in the list is the weight of the edge between the node and neighbor. So, the number of occurrences of each of the neighbors is calculated and is stored in map named neighbor2weight. The key to this map is the neighbor and the value to this map will be the weight of the edge between the node and the neighbor. We create another map named

weight2neighbors using this map where key is weight of the edge between node and neighbor and value is the list of neighbors who are connected to the node with that weight.[1]

Algorithm 9. Second map phase of game 3.

```
class MAPPER
    method MAP(lineNumber, Edge)
        index = Edge.find(":")
        node = Edge.substring(0,index)
        listOfNeighbors = Edge.substring(index+1,end)
        EMIT(node, listOfNeighbors)
```

Algorithm 10. Third map-reduce phase of game 3.

```
class MAPPER
    method MAP(lineNumber, Edge)
        index = Edge.find(":")
        node = Edge.substring(0,index)
        listOfNeighbors = Edge.substring(index+1,end)
        EMIT(node, listOfNeighbors)

class REDUCER
    method REDUCE(node, list(extNeighbors))
        extNeighbors = set(list(extNeighbors))
        extDegree = length(extNeighbors)
        marginalContribution = 1/(1+extDegree)
        EMIT(node, marginalContribution)
        for all extNeighbor ∈ extNeighbors do
            EMIT(extNeighbor, marginalContribution)
        end for
```

Two hop neighbors are the neighbors who are reachable in at most two hops. Let A and B be two nodes which are two hops away i.e., A and B are not connected directly but connected through another node. Let the node through which A and B are connected be C. Now, A and B are one hop neighbors of C and similarly all the pair of nodes which are two hops away will have a common neighbor from which both of them will be one hop away. So, each node in the list of neighbors (input to the reducer of this stage) are two hops away from every other node in the same list of neighbors. Also, some of the nodes which are two hops away might as well be connected by a single edge. In this case, the shortest distance between them is 1 and not 2. So, whenever a reducer encounters a neighbor having two different values as weight for a same edge, it always has to choose the least value. Also, the edges may have weights. So, the reducer also has to check whether the sum of weights of edges is less than the d_{cutoff}. The reducer in this phase essentially finds the neighbors which are one hop more than what it has received as input. In the current map-reduce phase, reducer has received the neighbors which are one hops away and has found the neighbors which are two hops away. This can be extended further for higher hops in the same way.

The reducer constructs a neighborString which contains neighbors that one more hop away and whose path length is less than or equal to the d_{cutoff}. So, for every neighbor in the list of

[1] Map(a→b) used in the algorithms of this article represents the mapping from a to b. Here, a is the key and b is the value. The key is always unique and it can have multiple values.

neighbors, the reducer writes the neighbor as key and a neighborString as value depending on the weight of the neighbor to the node (input to the reducer of this phase).

The second map-reduce phase of this algorithm makes the neighborhood grow by one more hop. The mapper in this phase is given in algorithm 9. It reads the output of the previous phase and breaks the line into (key,value) pairs and sends it to the reducer. The reducer in this phase is essentially the same as that of the previous map-reduce phase. This map-reduce phase is run iteratively until all the neighbors which are within d_{cutoff} away are visited.

The third map-reduce phase of this algorithm is given in algorithm 10. The mapper in this phase reads the input from the file and splits them into (key, value) pairs. The reducer in this stage receives nodes as keys and their extNeighbors as values. We calculate the extDegree of nodes by removing the duplicates and then counting the number of extNeighbors. Once the extDegree is calculated, marginal contribution is calculated by the formula $\frac{1}{1+extDegree}$. The reducer writes the marginal contributions to the output similar to game 1.

The fourth and the final map-reduce phase of this algorithm is essentially the same as that of the second phase of game 1 (algorithm 4). The mapper in this phase does not do any computation. The reducer in this phase gets nodes as keys and the marginal contributions of all their neighbors as values. It aggregates the marginal contributions of all the neighbors of a node to obtain the centrality of the node.

We note that game 3 algorithm can be applied on weighted networks also. The sum of weights on the path between node u and v is calculated as $Distance(u)$ from v which will be compared with d_{cutoff} to find whether node u can be included in the extended neighborhood of node v or not. In the unweighted networks, weights on the edges are assumed to be 1.

3.5 *Game 4: Number of agents in the network*

The fourth game theoretic centrality algorithm computes centrality by considering the fringe as the set of all nodes in the network. Algorithm 11 gives the steps to calculate this centrality. It has a running time of $O(VE + V^2 log(v))$.

Algorithm 11. Game 4.

Input: Weighted graph $G(V,E)$
Output: Centrality values of all nodes

```
for all v ∈ V do
    (Distances, Nodes) ← Dijkstra(v, G)
    sum = 0
    prevDistance = -1
    prevContribution = -1
    for all index ∈ |V-1| to 1 do
        if Distances(index) = prevDistance then
            contribution = prevContribution
        else
            contribution = f(D(index))/(1+index) - sum
        end if
        centrality[w(index)] += contribution
        sum += f(D(index))/(index(1+index))
        prevDistance = Distances(index)
        prevContribution = contribution
    end for
    centrality(v) += f(0) - sum
end for
```

From algorithm 11, we observe that in order to compute the centrality of a node, we need to find the extNeighbors and the distance to the extNeighbors for every node. This algorithm is an extension of game 3. The intuition behind this algorithm is that the power of influence decreases as the distance increases. Generally, the extNeighbors are all the nodes in the network for this game. But the contribution of neighbors who are farther away is not highly significant and so in our parallelization, we have fixed the d_{cutoff} to be 2. We parallelize this algorithm using four map-reduce phases.

The first and second phase of this algorithm is essentially the same as game 3 (algorithms 8 and 9). In these phases, neighbors of each of the nodes are found. Each map-reduce phase extends the neighborhood by one hop. This process is repeated iteratively until there is no more unvisited neighbors within the d_{cutoff}.

The third map-reduce phase of this algorithm is given in algorithm 12. The mapper in this phase reads the input from the file and splits them into (key, value) pairs such that every node in the graph is the key and the list of neighbors and weights as values. The weight here represents the distance from the node to the neighbor. Once the neighbors and weights are known, the marginal contributions are calculated according to the lines 3 to 17 of algorithm 11. The reducer writes the nodes and the marginal contributions to the output.

The fourth and the final map-reduce phase of this algorithm is essentially the same as that of the second phase of game 1 (algorithm 4) which aggregates the marginal contributions to obtain the centrality.

Algorithm 12. Third map-reduce phase of game 4.

```
class MAPPER
    method MAP(lineNumber, Edge)
        index = Edge.find(":")
        node = Edge.substring(0,index)
        listOfNeighbors = Edge.substring(index+1,end)
        EMIT(node, listOfNeighbors)

class REDUCER
    method REDUCE(node, list(neighbors,distances))
        index = length(neighbors)
        sum = 0
        prevDistance = -1
        prevContribution = -1
        for all neighbor ∈ neighbors do
            if distances(index) == prevDistance then
                contribution = prevContribution
            else
                contribution = f(D(index))/(1+index) - sum
            end if
            EMIT(neighbor, contribution)
            sum += f(D(index))/(index(1+index))
            prevDistance = Distances(index)
            prevContribution = contribution
            index = index - 1
        end for
        contribution = f(0) - sum
        EMIT(node, contribution)
```

3.6 *Game 5: Number of agents with* $\sum$*(weights inside C)* $\geq W_{cutoff}$*(agent)*

The fifth game theoretic centrality algorithm computes centrality by considering the fringe as the set of all nodes whose agent specific threshold is less than the sum of influences on the node by the nodes who are already in the coalition. Algorithm 13 gives the steps to calculate this centrality. It has a running time of $O(V + E^2)$.

From algorithm 13, we observe that in order to compute the centrality of a node, we need to find the degree of the node and its neighbors. This algorithm is an extension of game 2 (algorithm 5) for weighted networks. The intuition behind this algorithm is that every node will be influenced only when the sum of weights to all the active neighbors is greater than the cut-off of the node. Let α_v be the sum of weights of edges to all the neighbors of node v and β_v be the sum of squares of weights of edges to all the neighbors of v. Then, the results of the analysis done in Tomasz Michalak *et al* (2013) for this algorithm are as follows:

Algorithm 13. Game 5.

Input: Weighted graph $G(V, E)$
Output: Centrality values of all nodes

for all $v \in V$ **do**
  $W_{\text{cutoff}}(v)$ = Random(1,degree(v)+1)
end for
for all $v \in V$ **do**
  centrality(v) = 0
  for all m in 0 to $deg(v)$ **do**
    $\mu = \mu(X_m^{vv})$
    $\sigma = \sigma(X_m^{vv})$
    $p = Pr\{\mathcal{N}(\mu, \sigma^2) < W_{\text{cutoff}}(v)\}$
    centrality(v) += $\frac{p}{1+deg(v)}$
  end for
  for all $u \in$ Neighbors(v) **do**
    p = 0
    for all m in 0 to $deg(v) - 1$ **do**
      $\mu = \mu(X_m^{uv})$
      $\sigma = \sigma(X_m^{uv})$
      $Z = Z_m^{uv}$
      p += z $\frac{deg(u)-m}{deg(u)(deg(u)+1)}$
    end for
    centrality(v) += p
  end for
end for

$$\mu(X_m^{vv}) = \frac{m}{deg(v)}\alpha_v$$

$$\sigma(X_m^{vv}) = \frac{m(deg(v)-m)}{deg(v)(deg(v)-1)}\left(\beta_v - \frac{\alpha_v^2}{deg(v)}\right)$$

$$\mu(X_m^{uv}) = \frac{m}{deg(v)-1}(\alpha_v - w(u,v))$$

$$\sigma(X_m^{uv}) = \frac{m(deg(v)-1-m)}{(deg(v)-1)(deg(v)-2)}\left(\beta_v - w(u,v)^2 - \frac{(\alpha_v - w(u,v))^2}{deg(v)-1}\right)$$

$$Z_m^{uv} = \frac{1}{2}\left[erf\left(\frac{W_{\text{cutoff}}(v) - \mu(X_m^{uv})}{\sqrt{2}\sigma(X_m^{uv})}\right) - erf\left(\frac{W_{\text{cutoff}}(v) - w(u,v) - \mu(X_m^{uv})}{\sqrt{2}\sigma(X_m^{uv})}\right)\right]$$

We parallelize this algorithm using two map-reduce phases.

The first map-reduce phase of this algorithm is given in algorithm 14. The only difference between the first phase of game 1 and game 5 algorithms is the way in which the marginal contributions are calculated. So, the mapper does the same job as algorithm 3. The input to the reducer is nodes and their neighbors. Since, weighted graphs will be the input for game 5, we take the count of the occurrences of neighbor which will give the weight of the edge between the node and the neighbor. Marginal contribution of a node to itself is calculated as per the lines 5–11 of algorithm 13 in lines 5–6 of the reduce phase of algorithm 14. Marginal contribution of a node to its neighbor is calculated as per the lines 12–21 of algorithm 13 in lines 7–10 of the reduce phase of algorithm 14. Once the marginal contributions are calculated, the reducer writes the nodes and the corresponding marginal contributions to the output. The second map-reduce phase of this algorithm is essentially the same as that of the second phase of game 1 (algorithm 4) which aggregates the marginal contributions to obtain the centrality.

Algorithm 14. First map-reduce phase of game 5.

```
class MAPPER
    method MAP(lineNumber, edge)
        (nodeA, nodeB) = edge.split()
        EMIT(nodeA, nodeB)
        EMIT(nodeB, nodeA)

class REDUCER
    method REDUCE(node, list(neighbors))
        (neighbor,edgeWeight) ← neighbors
        degree = length(neighbor)
        weightedDegree = sum(edgeWeight)
        marginalContribution = contribution(threshold)
        EMIT(node, marginalContribution)
        for all neighbor ∈ neighbors do
            marginalContribution = contribution(threshold, edgeWeight)
            EMIT(neighbor, marginalContribution)
        end for
```

4. Experimental results

The experiments in this section are designed to show (i) that the game-theoretic centrality measures perform on par with the greedy algorithm in the context of information diffusion. (ii) The scalability of the parallelized algorithms with respective to the size of the input graph. (iii) The scalability of the algorithms with respect to the number of machines in the hadoop cluster.

We have verified the results of the parallelized algorithms on several small synthetic networks generated using the E-R model and found them to be correct, in that they returned the analytically computed Shapley values for all the nodes.

4.1 *Cascade experiment*

Influence spread was explored in the literature in two different contexts. One is top k problem and the other is λ coverage problem. In the top k problem, a set of k nodes that will maximize the influence spread in the network has to be selected. In the λ coverage problem, the minimum value of k that is needed to obtain the desired influence spread has to be found. We have chosen the context of top k problem for our experiment. We did the cascade experiment to find the influence spread of the top k nodes given by five game theory based centrality algorithms and the greedy algorithm. We have used the linear threshold model to find the influence spread.

Influence model: In this model, every node can influence the neighbor nodes and every neighbor node can influence the node. Consider the edge (u, v) from node u to node v in a graph. Let d_v be the sum of weights of incoming edges of node v. In an unweighted network, d_v becomes degree of node v. Let $C_{u,v}$ be the weight of the edge from u to v. Then, node u has an influence of $\frac{C_{u,v}}{d_v}$ on node v. Note that the sum of influences on any node is 1. A node can influence its neighbors only if it is influenced already. There will be initial set of influencers who are assumed to be influenced by external forces. This model assumes that every node in graph has a threshold associated with it. A node is said to be influenced if sum of influences on that node by its neighbors is greater than threshold of the node. Neighbors of the newly influenced node might get influenced based on the above criteria. The influence spread stops when there is no more node in the graph which could be influenced. The threshold of the nodes assumed by this model cannot be determined in the real world scenarios. So, a number of different sets of thresholds are sampled from an appropriate probability distribution and the influence spread is run for every set of threshold. The average of all these runs gives the number of nodes influenced. Typically, several different thresholds are sampled and run to cancel out the effect of threshold.

Greedy algorithm: Consider a network with n nodes. Greedy algorithm runs the cascade spread n times with every node as a initial influencer. The node which is able to influence most number of nodes in the network is considered to be the top 1 influencer. Then it considers the rest of $n - 1$ nodes and runs the cascade experiment n-1 times with top 1 influencer and every one of the n-1 nodes as initial influencers. The node which when combined with the top 1 influencer influences most number of nodes is added to the set of top influencers and thus top 2 influencers are formed. This process is repeated k times to find the top k influencers of the network. The problem of finding the exact top k nodes is hard. Greedy algorithm is the best approximation for this problem (Kempe *et al* 2003).

We ran the cascade experiment in the collaboration network data. The network data can be obtained from http://snap.stanford.edu/data/ca-GrQc.html. This network is between collaborations of authors. Nodes in the network correspond to the authors. If an author u had collaborated with an author v in at least one paper, then an undirected edge is established between nodes u and v in the network. This network consists of 5242 nodes and 14496 edges. Figure 4 shows the plot of the number of nodes influenced against the number of initial influencers in this network. We observe that greedy algorithm performs well when the number of initial influencers is very less and the performance of game theoretic centrality algorithms increases as the number of initial influencers is increased. At $k = 30$, Game 4 and Game 5 perform marginally better than the greedy algorithm while Game 1 performs close to the greedy algorithm. Though Game 2 and Game 3 (dotted lines in figure 4) algorithms give a slightly lesser performance than greedy algorithm, the difference is not enormous. We also note that the greedy algorithm does not scale well in terms of size of the network size as well as the number of initial influencers. Detailed analysis of the game theoretic centrality algorithms is done by Tomasz Michalak *et al* (2013) and Narayanam & Narahari (2011).

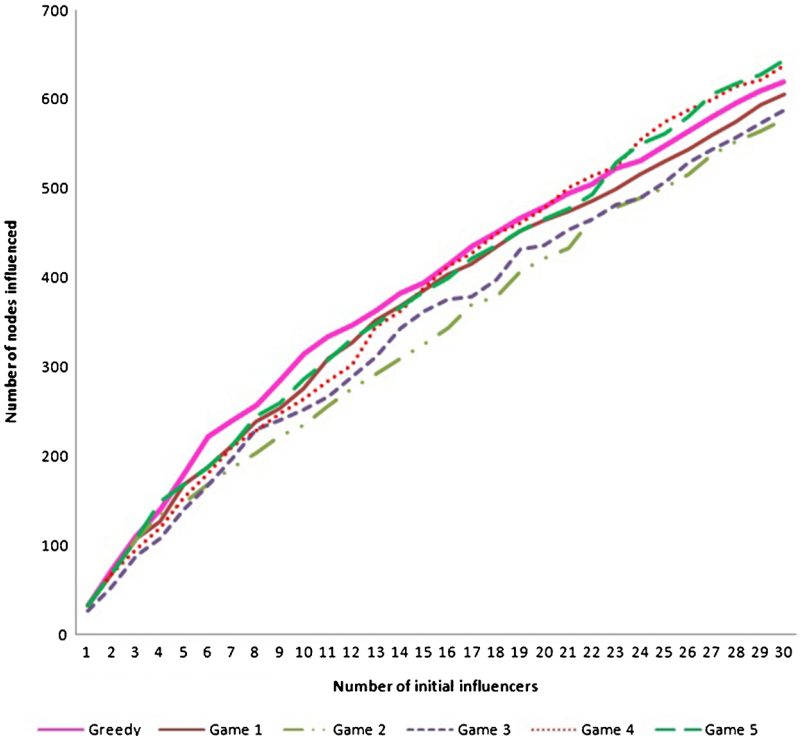

Figure 4. Result of cascade experiment in collaboration network.

4.2 *Scalability experiments*

We generated synthetic networks to analyse the scalability of our algorithms. The synthetic networks were generated using the Barabasi–Albert model and Erdos–Renyi model.

Barabasi–Albert model: This model starts with small seed graph. This seed graph needs to be connected i.e., there should be a path from every node in the graph to every other node in the graph, to make sure that the final graph will be connected. The average degree of nodes in the network can be calculated by the formula,

$$\text{Average degree} = \frac{2 * \text{ Number of edges in the graph}}{\text{Number of nodes in the graph}}.$$

Let x denote half of the average degree. The number of nodes in the seed graph should be a little higher than x. For every other node in the network, x nodes have to be chosen from the seed graph and all the x nodes should be connected to this new node. The seed graph grows with the addition of every node and once every node gets added to the seed graph, it becomes the final graph. Note that the probability with which the x nodes in the seed graph gets selected is proportional to the degree of the nodes of the seed graph i.e., higher the degree, more is the probability of getting selected. So, the resulting graph will have few nodes with high degree and many nodes with average degree.

Erdos–Renyi(E–R) model: This model generates network with the probability of existence of the edge. Depending on the density of the edges needed in the graph, the probability of existence of an edge can be calculated as follows.

$$\text{Probability of existence an edge, } P = \frac{2 * \text{ Number of edges in the graph}}{\text{Number of possible edges in the graph}}.$$

Every edge in the graph is included with probability P and discarded otherwise. We observe that the graph will have no edges if P is zero and all the edges if P is 1.

Scalability with respect to input size: We ran our parallelized algorithms in a hadoop cluster having 10 machines with each machine having 8 cores. We have varied the size of the input graph and have calculated the running time of these parallelized algorithms. Tables 1–6 show the running time of parallelized version of the Game 1, Game 2, Game 3, Game 4 and Game 5 algorithms in seconds for different input sizes and densities of the graphs.

Table 1. Running time of game 1 on E-R graphs of different densities in seconds.

Network Size	Density of edges			
(# of edges)	0.01	0.001	0.0001	0.00001
10^3	28	28	28	29
10^4	31	29	29	29
10^5	31	31	31	33
10^6	45	46	46	45
10^7	84	85	79	74

Table 2. Running time of game 2 on E-R graphs of different densities in seconds.

Network Size	Density of edges			
(# of edges)	0.01	0.001	0.0001	0.00001
10^3	29	29	30	28
10^4	30	28	29	29
10^5	32	29	31	33
10^6	41	46	45	44
10^7	83	80	81	74

Table 3. Running time of game 3 on E-R graphs of different densities in seconds.

Network Size	Density of edges			
(# of edges)	0.01	0.001	0.0001	0.00001
10^3	43	42	41	43
10^4	43	45	44	44
10^5	49	46	49	46
10^6	109	89	70	70
10^7	1616	913	301	167

Table 4. Running time of game 4 on E-R graphs of different densities in seconds.

Network Size (# of edges)	Density of edges			
	0.01	0.001	0.0001	0.00001
10^3	42	43	44	42
10^4	46	43	45	45
10^5	51	50	45	46
10^6	110	89	70	71
10^7	1735	886	312	176

Table 5. Running time of game 5 on E-R graphs of different densities in seconds.

Network Size (# of edges)	Density of edges			
	0.01	0.001	0.0001	0.00001
10^3	30	28	29	28
10^4	30	30	30	29
10^5	33	32	30	32
10^6	45	45	46	43
10^7	86	114	79	83

Table 6. Running time of all the game theoretic algorithms on Barabasi Albert graphs of density 0.1 in seconds.

Network Size	Game 1	Game 2	Game 3	Game 4	Game 5
10^3	33	30	45	46	30
10^4	31	29	47	48	30
10^5	31	30	93	90	39
10^6	60	59	146	141	67
10^7	73	79	5128	5354	128

Scalability with respect to number of CPUs in synthetic networks: We ran our parallelized algorithms in a hadoop cluster by varying the number of machines in the cluster for a fixed input size. We chose the number of edges in the input graph to be 1 million for running game 3 and game 4 while we chose it to be 10 million for game 1, game 2 and game 5. Tables 7 and 8 show the running time of the algorithms when the number of machines in the cluster is varied. Each machine in our hadoop cluster had an 8 core CPU. So, 64 mappers or reducers can be run at a time.
Scalability with respect to number of CPUs in real world graphs: We obtained the real world networks from Stanford large network data. The size of the network is given in table 9.

Tables 10–12 show the results on the Amazon, DBLP and Youtube networks. Increasing the number of CPUs did not have much effect on the Amazon and DBLP network. On the other hand, a performance improvement is seen in the Youtube network for game 3 and game 4 algorithms when we use multiple CPUs instead of 1 CPU. In the Amazon and DBLP network, for game 1 and game 2, one CPU performs better than multiple CPUs. The reason is that when we force hadoop to all the CPUs, when they are not required, communication overhead becomes a bottleneck.

Table 7. Running time of game 1, game 2 and game 5 algorithms in seconds on a 10 million edge network.

Number of machines	Game 1	Game 2	Game 5
1	101	103	167
2	94	82	113
3	92	85	114
4	90	84	102
5	87	84	97
6	84	84	97
7	83	83	96
8	82	84	96

Table 8. Running time of game 3 and game 4 algorithms in seconds on a 1 million edge network.

Number of machines	Game 3	Game 4
1	632	606
2	289	313
3	206	213
4	176	179
5	166	172
6	146	144
7	137	141
8	146	141

Table 9. Networks with ground-truth communities.

Networks	Nodes	Edges
Youtube	1,134,890	2,987,624
DBLP	317,080	1,049,866
Amazon	334,863	925,872

Table 10. Running time of all the game theoretic algorithms on Amazon network in seconds.

Number of machines	Game 1	Game 2	Game 3	Game 4	Game 5
1	48	48	96	97	50
2	58	60	98	101	60
3	60	60	97	102	58
4	62	62	95	98	59
5	62	62	99	100	59
6	62	61	97	97	60
7	62	62	97	99	62
8	62	62	98	101	61

Table 11. Running time of all the game theoretic algorithms on DBLP network in seconds.

Network size	Game 1	Game 2	Game 3	Game 4	Game 5
1	49	49	124	128	50
2	57	60	108	115	60
3	58	60	108	106	62
4	58	60	107	104	61
5	61	62	107	104	61
6	62	60	102	101	63
7	62	60	100	104	61
8	61	60	101	104	61

Table 12. Running time of all the game theoretic algorithms on Youtube network in seconds.

Network size	Game 1	Game 2	Game 3	Game 4	Game 5
1	66	67	11276	12223	125
2	77	70	6720	7435	104
3	75	76	4530	4718	101
4	78	70	2921	3037	94
5	73	74	2221	2221	93
6	71	73	1766	1895	94
7	70	69	1602	1750	93
8	70	71	1580	1694	93

From the above experiments, we observe the following:

- We can process the networks of million edges in few seconds using the parallelization techniques even if the algorithm has a quadratic time complexity (e.g., game 3 and game 4).
- Game 1, game 2 and game 5 are highly scalable as these algorithm take only few seconds for running on network sizes which are as large as few millions of edges.
- The running time of the algorithms decreases as the density of edges in the graph decreases. This is due to the fact that every node in the sparse graph will have less number of neighbors when compared to the nodes in the dense graphs.
- Performance cannot be increased beyond a point by adding more number of CPUs for a given input size which can be inferred from the results of game 1 and game 2 algorithms on Amazon and DBLP networks.
- The running time reduces as the number of machines in the cluster is increased which can be inferred from the results of game 3 and game 4 algorithms on the Youtube network.

5. Conclusions

In this work, we have presented parallel versions of several Shapley value based centrality algorithms and have shown their scaling behavior on networks with a few million edges. We were able to process most of these large networks in few seconds. We have also presented an elegant

way to construct the adjacency list from the edge list representation of a graph. This allows us to easily handle large scale interaction data, where the data is generated one edge at a time and solves the basic problem of finding one hop neighbors in graph in a parallelized fashion. We achieved this scaling with only 64 cores. By employing more cores we can process large graphs of the orders of magnitude common in online social networks, without any modification in the algorithms proposed in this work. Currently we are conducting empirical studies on large real interaction network data in collaboration with a leading telecommunications company. In summary, this work has opened up the possibility of using complex information diffusion centrality models in large networks. This was not possible earlier due to the computational complexity and the inherent serial nature of the popular measures of centrality.

References

Apache Software Foundation 2011 Apache hadoop. http://hadoop.apache.org/

Asavathiratham C, Roy S, Lesieutre B and Verghese G 2001 The influence model. *Control Systems, IEEE*

Bass Frank M 1969 A new product growth for model consumer durables. *Manag. Sci.*

Bonacich Phillip 1987 Power and centrality: A family of measures. *Am. J. Soc.*1170–1182

Borthakur D 2007 The hadoop distributed file system: Architecture and design. *Hadoop Project Website*

Brown Jacqueline J and Reingen Peter H 1987 Social ties and word-of-mouth referral behavior. *J. Consum. Res.*

Carrington P J, Scott J and Wasserman S 2005 *Models and methods in social network analysis*. Cambridge University Press

Dean Jeffrey and Ghemawat Sanjay 2004 Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation*

Domingos P and Richardson M 2001 Mining the network value of customers. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*

Gibbons Robert 1992 *A primer in game theory*. Harvester Wheatsheaf

Hilbert Martin and Löpez Priscila 2011 The world's technological capacity to store, communicate, and compute information. *Science* 60–65

Kempe D, Kleinberg J and Tardos E 2003 Maximizing the spread of influence through a social network. In: *KDD*, 137–146

Lerman Kristina and Ghosh Rumi 2010 Information contagion: an empirical study of the spread of news on digg and twitter social networks. *Computing Research Repository*

Linyuan L and Tao Zhou 2011 Link prediction in complex networks: A survey. *Physica A* 1150–1170

McKinsey Global Institute 2011 Big data: The next frontier for innovation, competition, and productivity. Technical report

Tomasz Michalak, Karthik V Aadithya, Szczepanski L, Balaraman Ravindran and Nicholas R Jennings 2013 Efficient computation of the Shapley value for game-theoretic network centrality. *J. Artif. Intell. Res.* 607–620

Narayanam Ramasuri and Narahari Y 2008 Determining the top-k nodes in social networks using the Shapley value, 1509–1512. IFAAMAS

Narayanam Ramasuri and Narahari Yadati 2011 A Shapley value-based approach to discover influential nodes in social networks. *IEEE Trans. Autom. Sci. Eng.* 130–147

Richardson Matt and Domingos Pedro 2002 Mining knowledge-sharing sites for viral marketing. In *KDD*