Peer-to-Peer Systems Final Project

Andrea Maggiordomo – mggndr89[at]gmail.com
Master degree in Computer Science, Università di Pisa

2016

1 Introduction

The task of the project was to define Deccen, a decentralized algorithm to compute stress centrality indices in a network based on the ideas introduced in [8], and study its behavior using the PeerSim [10] simulator. Additionally – given the high cost of the algorithm – approximation techniques could be proposed in order to reduce the communication overhead and memory requirements, and the algorithm could be extended to also compute the closeness and betweenness centrality indices.

This report defines a version of Deccen capable of computing all the three centrality measures mentioned. It also defines a different decentralized algorithm based on the previous works of Brandes [1, 2], and Eppstein and Wang [3], to approximate centrality values by sampling a limited amount of network nodes. This second algorithm, referred to as Multi-BFS, has a significantly smaller communication overhead.

This document is organized as follows: in section 2 some preliminary definitions are given, while Deccen is defined in section 3 and Multi-BFS is defined in section 4. Experimental results are shown in section 5: the aims of the experiments were to compare the performance of the two algorithms, and to evaluate the quality of the estimates obtained with Multi-BFS. Follows a small overview of the most relevant choices made in the development of the project code. The proofs of some results introduced in section 4 are reported in the appendix.

2 Preliminary definitions and assumptions

The task is to compute centrality indices for a given undirected graph $G = (V, E)$, which is assumed to be connected. Unless otherwise stated, n denotes the number of nodes $|V|$ and m the number of edges $|E|$. Each vertex $v \in V$ represents a network node with some given computational power, that can only communicate with its direct neighbors $N_v = \{u \in V : \{u, v\} \in E\}$. The terms *vertex* and *node* will be used interchangeably.

A *path* of *length* k from a source $s \in V$ to a destination $t \in V$ is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $s = v_0$, $t = v_k$ and $\{v_{i-1}, v_i\} \in E$ for $i = 1, 2, \ldots, k$. The *distance* $d(u,v)$ between two vertices is the length of the shortest path that connects them (with $d(u,u) = 0$) while the diameter Δ is the maximum distance between any pair of vertices. Note that $d(u,v) = d(v,u)$ since the graph is undirected. A vertex v is a *predecessor* of w with respect to s if $\{v,w\} \in E$ and $d(s,v) + 1 = d(s,w)$. The *predecessor set* $P_s(w)$ of a vertex w is the set of all predecessors of w with respect to s.

The number of different shortest paths that connect two vertices $s, t \in V$ is denoted by σ_{st}, while the quantity $\sigma_{st}(v)$ is the number of shortest paths between s and t that pass through v (this means that if $v = s$ or $v = t$ then $\sigma_{st}(v)$ is always zero).

The centrality indices relevant to this document are the following:

Closeness centrality. The *closeness* centrality $C_C(v)$ of a vertex $v \in V$ is

$$C_C(v) = \frac{\sum_{u \in V} d(u,v)}{n-1} \tag{1}$$

Stress centrality. The *stress* centrality $S_C(v)$ of a vertex $v \in V$ is

$$S_C(v) = \sum_{s \in V} \sum_{t \in V} \sigma_{st}(v) \tag{2}$$

Betweenness centrality. The *betweenness* centrality $B_C(v)$ of a vertex $v \in V$ is

$$B_C(v) = \sum_{s \in V} \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3}$$

The algorithms described in this report all assume an underlying synchronous communication model where the computation evolves in steps: at each step all the network nodes perform their computations independently and autonomously, and the messages they send at step t are delivered to the destinations and processed at step $t + 1$.

3 The DECCEN algorithm

The specification of DECCEN is based on the algorithmic scheme outlined in Lehmann and Kaufmann [8]. Initially, each node broadcasts itself on the network. Exploiting the synchronous model, after k steps any node $t \in V$ will know all the nodes $s \in V$ such that $d(s,t) = k$ and the number of shortest paths σ_{st} that links it to each of them. This information is stored locally and also reported in broadcast to allow other nodes $v \in V$ to compute the quantity $\sigma_{st}(u)$ necessary to compute the betweenness and stress centrality indices. The value $\sigma_{st}(u)$ can be determined by exploiting the following lemma.

Lemma (Bellman conditions). *A node $v \in V$ lies on a shortest path from $s \in V$ to $t \in V$ if and only if $d(s,t) = d(v,s) + d(v,t)$.*

The synchronous model ensures that if a node v lies on a shortest path between s and t and receives a report for such a pair, it has already computed σ_{vs} and σ_{vt}. Then, according to the above conditions $\sigma_{st}(v) = \sigma_{vs} \cdot \sigma_{vt}$.

3.1 Message types

DISCOVERY$\langle s, u, d, \sigma_{su} \rangle$ These messages are used to track distances and the number of shortest paths from a specific origin. They contain the source $s \in V$ of the broadcast, the sender $u \in V$, the distance $d = d(s,u)$ of u from s and the number of shortest path that connect u to s.

REPORT$\langle (s,t), \sigma_{st}, d_{st} \rangle$ These messages are broadcast by t after having determined the number of shortest paths to s and the distance from it.

3.2 Node state

Each node v maintains three accumulators C_C, C_B and C_S for closeness, betweenness and stress centrality (the closeness centrality will be retrieved as $1/C_C$), a set R of node pairs for which a REPORT message has already been received, and two dictionaries D and S that associate each node $s \in V$ with the discovered distance $d(s,v)$ and the number of shortest paths σ_{sv} respectively.

3.3 Protocol initialization

Each node $v \in V$ initializes the accumulators C_C, C_B and C_S to 0, the set R to the empty set, the dictionary D so that it only contains the entry $(v, 0)$ and S so that it only contains the entry $(v, 1)$. Furthermore, it sends to all its neighbors a DISCOVERY$\langle v, v, 0, 1 \rangle$ message.

3.4 Step actions

The processing performed by each node v at each step is the following:

1. All the DISCOVERY messages having a source s for which the dictionary D contain no mapping are grouped together. (These will be the nodes "discovered" at this step).

2. Each group of messages is processed independently. For each group, let s be the source and d be the distance of all the DISCOVERY$\langle s, u, d, \sigma_{su} \rangle$ messages in it (these will be the same for all the messages), then:

   2.1. Add the entry $(s, d+1)$ to the dictionary D, so that $D[s] = d + 1$.

2.2. Let $\sigma_{sv} = \sum_u \sigma_{su}$ and add the entry (s, σ_{sv}) to S. (The number of shortest paths from s to v is the sum of the number of shortest paths from s to all the predecessors of v).

2.3. Send a REPORT$\langle (s, v), \sigma_{sv}, d + 1 \rangle$ message to each neighbor node.

3. For each REPORT$\langle (s, t), \sigma_{st}, d_{st} \rangle$ message such that $(s, t) \notin R$:

3.1. If $s = v$ then $C_C \leftarrow C_C + d_{st}$.

3.2. If $d_{st} = D[s] + D[t]$ let $\sigma_{st}(v) = \sigma_{sv} \cdot \sigma_{tv}$, then $C_B \leftarrow \frac{\sigma_{st}(v)}{\sigma_{st}}$ and $C_S \leftarrow \sigma_{st}(v)$.

3.3. Add the pair (s, t) to the set R, and forward the REPORT message to all the neighbors.

3.5 Cost analysis

The broadcast of a DISCOVERY or a REPORT requires $O(m)$ messages. Since each nodes starts a DISCOVERY and generates $n-1$ reports the total number of messages exchanged is $O(nm + n^2m)$, with REPORT messages inducing the dominant factor n^2m. An optimization that can be performed by a node v at step 3.3 is to avoid the propagation of a (s, t) REPORT if $\sigma_{st}(v) = 0$. In this case the report is irrelevant to v and any neighbor that may have needed it will have received it earlier from the broadcast of nodes that lie on shortest st-paths.

In terms of memory consumption, each node will add $O(n)$ entries each of the two dictionaries and $O(n^2)$ pairs to the set R.

4 Approximation of centrality indices

The main issues with DECCEN are the high cost in terms of number of messages exchanged and the local requirement of a quadratic space data structure, which make its use impractical in networks of reasonable size.

Eppstein and Wang introduced in [3] an approximation algorithm for the computation of closeness centrality. Their approach is to sample from a subset of nodes the contributions to the closeness of all the nodes in the network, and extrapolate an approximation from them; the contribution of a node s to any v is taken to be the distance $d(s, v)$ and is computed by solving a Single-Source-Shortest-Path problem with s as source. Brandes in [2] extended this scheme to the approximation of betweenness centrality: his method relies on reformulating the betweenness in terms of contributions from other nodes and employs an augmented SSSP to compute those contributions recursively, as he originally proposed in [1]. This approach can be applied in a decentralized way (by adapting the SSSP problem, which in this case becomes a decentralized Breadth-First Search) and extended to the estimation of stress centrality, leading to the definition of another decentralized algorithm for the computation (and estimation) of centrality indices.

4.1 Reformulating centrality indices

The first step is to write each centrality index of a node v as a sum of terms where each term denotes the contribution of a second node s (referred to as the *source* of the contribution) to the index value.

Contribution of a source to Closeness centrality As already mentioned, the contribution of a source node s to the closeness centrality of v is the distance $d(s, v)$. The closeness of a node is simply the reciprocal of the sum of the contributions of all the other nodes to its centrality.

Contribution of a source to Betweenness centrality The contribution of a source s to the betweenness centrality of v is the *dependency* of s on v introduced in [1]:

$$\delta(s|v) = \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{4}$$

that allows to rewrite the betweenness centrality of v as

$$BC(v) = \sum_{s \in V} \delta(s|v). \tag{5}$$

Contribution of a source to Stress centrality The contribution to stress centrality is analogous to the contribution to betweenness centrality:

$$\sigma(s|v) = \sum_{t \in V} \sigma_{st}(v), \tag{6}$$

and the stress centrality of v is rewritten as

$$SC(v) = \sum_{s \in V} \sigma(s|v). \tag{7}$$

4.2 Computing contributions from a single source

A decentralized Breadth First Search can be adapted to compute the contribution of a source to any of the three centrality indices considered.

The closeness centrality contribution is simply the distance from the source to the node and it can be computed directly during the visit.

For betweenness centrality, Brandes proved in [1] that dependencies of a source obey a recursive relation expressed in terms of predecessors set:

Theorem 1 (Brandes, 2001). *The dependency of $s \in V$ on any $v \in V$ obeys*

$$\delta(s|v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta(s|w)). \tag{8}$$

For stress centrality contributions a similar relation holds (the proof is reported in the appendix):

Theorem 2. *The stress centrality contribution of $s \in V$ on any $v \in V$ obeys*

$$\sigma(s|v) = \sum_{w: v \in P_s(w)} \sigma_{sv} \cdot \left(1 + \frac{\sigma(s|w)}{\sigma_{sw}}\right). \tag{9}$$

The predecessor sets can be easily discovered by the BFS algorithm during the network exploration, while contributions are computed with a backward walk from the frontier of the BF-Tree to the source of the visit.

4.3 Random sampling of source nodes

To let the algorithm operate in a decentralized way, each node independently initiates a visit with a given probability p, which reflects the fraction of the network sampled. Even if the number k of samples is not known beforehand, all the nodes will be able to compute it by counting the number of visits in which they will take part.

The result of sampling from a source node v_i at a node u is modeled with the following random variables that rely on the contributions of v_i to the centrality values of u:

$$X_i(u) = \frac{n}{n-1} \cdot d(v_i, u), \qquad Y_i(u) = n \cdot \delta(v_i|u), \qquad Z_i(u) = n \cdot \sigma(v_i|u).$$

These variables are then used to approximate centrality indices with the following estimators:

$$\widetilde{C}_C(u) = \sum_{i=1}^{k} \frac{X_i(u)}{k}, \qquad \widetilde{C}_B(u) = \sum_{i=1}^{k} \frac{Y_i(u)}{k}, \qquad \widetilde{C}_S(u) = \sum_{i=1}^{k} \frac{Z_i(u)}{k}.$$

The derivations for the next theorem – which ensures the estimators are unbiased – are reported in the appendix.

Theorem 3 (Unbiased estimators)**.** *The expected values of the centrality estimators are the actual centrality indices:*

(a) $\mathbf{E}[\widetilde{C}_C(u)] = \widehat{C}_C(u)$,

(b) $\mathbf{E}[\widetilde{C}_S(u)] = C_S(u)$,

(c) $\mathbf{E}[\widetilde{C}_B(u)] = C_B(u)$.

4.4 Multi-BFS algorithm specification

The only parameter of the algorithm is the probability p of a node becoming the source of a decentralized BFS and having its centrality contributions sampled at every location. The network size n is assumed to be known in advance to all the nodes, otherwise it can be easily computed during the backtracking phase of the augmented BFS and broadcast by one or more sources.

Note that if this algorithm is executed with $p = 1$ it is basically the decentralized version of the algorithm described in [1], and at the end of the execution the estimators will yield the exact centrality values.

4.4.1 Message types

DISCOVERY$\langle s, u, d, \sigma_{su} \rangle$ Messages of this type are used during the visit from a source to build the predecessors sets at each node. The fields are the source s of the visit, the sender u, the distance d of the sender to the source (that is, $d = d(s, u)$) and the number of shortest path from the source to the sender σ_{su}.

REPORT$\langle s, v, \delta(s|v), \sigma(s|v), \sigma_{sv} \rangle$ These messages are sent by a node v as part of the backward walk to inform its predecessors of the computed contributions and to allow them to compute their own by applying the recursive relations introduced in section 4.2.

4.4.2 Visit states

Visit states are parametric with respect to the discovery from a source $s \in V$. A node v is in state:

WAITING(s) if it has not yet received any DISCOVERY having s as source.

ACTIVE(s) if it has received one or more DISCOVERY messages with source s and has not yet computed the contributions of s to its centrality indices.

COMPLETED(s) if it has computed the contributions of s to its centrality indices and reported them back to each predecessor in $P_s(v)$ by sending appropriate REPORT messages.

4.4.3 Node state

Each node v maintains three centrality accumulators C_C, C_B and C_S like in DECCEN, and a counter k to track the number of sample nodes involved in the procedure. Furthermore, while a node is in state ACTIVE(s) when dealing with a visit from s it will need to partition the set of neighbors N_v in three subsets: the set of predecessors $P_s(v)$, the set of siblings $S_s(v)$ and the set of children $C_s(v)$, as well as track contributions of s to its centrality scores with three parametric accumulators $C_C^{(s)}$, $C_B^{(s)}$ and $C_S^{(s)}$.

4.4.4 Protocol initialization

Upon initialization, a node v clears its centrality accumulators and the counter k, and enters state WAITING(s) for all $s \in V$. Then, during the initial step it initiates a visit with probability p by entering state ACTIVE(v), setting $P_v(v) = \emptyset$ and sending a DISCOVERY$\langle v, v, 0, 1 \rangle$ to every neighbor.

4.4.5 Step actions

The actions performed by a node v at each step are the following:

1. The messages received at the current step are divided by type and then DISCOVERY messages are grouped by source.

2. For each group of DISCOVERY$\langle s, u, d, \sigma_{su}\rangle$ messages having s as a common source (at distance d), with v in state WAITING(s):

   2.1. Change state to ACTIVE(s) and initialize $C_C^{(s)} \leftarrow d+1$, $C_B^{(s)} \leftarrow 0$ and $C_S^{(s)} \leftarrow 0$; then collect all the senders u in the predecessor set $P_s(v)$ and compute $\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su}$.

   2.2. If $P_s(v) = N_v$ send a REPORT$\langle s, v, 0, 0, \sigma_{sv}\rangle$ message to all the predecessors $u \in P_s(v)$ and change state to COMPLETED(s).

   2.3. Otherwise, send to all $w \in N_v \setminus P_s(v)$ a DISCOVERY$\langle s, v, d+1, \sigma_{sv}\rangle$ message and change state to ACTIVE(s).

3. For each group of DISCOVERY$\langle s, u, d, \sigma_{su}\rangle$ messages having source s and distance d with v in state ACTIVE(s):

   3.1. Under the synchronous model assumption v can only receive such messages from nodes u such that $d(s,u) = d(s,v)$. Collect these nodes in the set $S_s(v)$ of the siblings of v with respect to s. (Note that these messages are received exactly one step after v has been contacted by its predecessors).

4. For each REPORT$\langle s, w, \delta(s|w), \sigma(s|w), \sigma_{sw}\rangle$ message received with v in state ACTIVE(s):

   4.1. Add w to the children set $C_s(v)$.

   4.2. Update $C_B^{(s)} \leftarrow C_B^{(s)} + \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta(s|w))$.

   4.3. Update $C_S^{(s)} \leftarrow C_S^{(s)} + \sigma_{sw} \cdot (1 + \frac{\sigma(s|w)}{\sigma_{sw}})$.

5. For any $s \in V$ with v is in state ACTIVE(s), if $P_s(v) \cup S_s(v) \cup C_s(v) = N_v$ then:

   5.1. If $s \neq v$ update the accumulators:
      - $C_C \leftarrow C_C + C_C^{(s)}$
      - $C_B \leftarrow C_B + C_B^{(s)}$
      - $C_S \leftarrow C_S + C_S^{(s)}$

      and send a REPORT$\langle s, v, \delta(s|v), \sigma(s|v), \sigma_{sv}\rangle$ message to all the predecessors contained in $P_s(v)$.

   5.2. Increment the counter k and change state to COMPLETED(s).

A node v can obtain the value of the estimators in the following way:

$$\widetilde{C}_C(v) = \frac{nC_C}{k(n-1)}, \qquad \widetilde{C}_B(v) = \frac{nC_B}{k}, \qquad \widetilde{C}_S(v) = \frac{nC_S}{k}.$$

4.5 Cost analysis

Each independent search requires $O(m)$ messages for the DISCOVERY phase and $O(m)$ messages to backtrack with REPORT messages. The parameter p determines the fraction of nodes pn that initiate a DISCOVERY search, so the total number of messages is $O(\lceil pn \rceil m)$.

In terms of memory consumption, note that for each $v \in V$ there are at most $\lceil pn \rceil$ other nodes s for which v is in state ACTIVE(s) and needs to partition its neighbor set N_v in the three subsets $P_s(v)$, $S_s(v)$ and $C_s(v)$, so the cost is $O(\lceil pn \rceil N_v)$.

5 Experiments

The aim of the simulations was to compare the number of messages required by the algorithms to complete the decentralized computation of the indices, and to measure the error of the estimates computed by MULTI-BFS with different values of the parameter p.

The networks used in the experiments (taken from the KONECT [7] repository) are the following:

`dolphins` This is a social network of bottlenose dolphins where nodes are dolphins and edges represent frequent associations. ($n = 62$, $m = 159$, source [9]).

`surf` This network that represents interpersonal contacts between windsurfers in southern California during the fall of 1986. ($n = 62$, $m = 336$, source [4]).

`macaques` This is a directed network that represent dominance behavior in a colony of 62 adult female Japanese macaques. The undirected version was obtained by forcing the edges to be symmetric. ($n = 62$, $m = 1167$, original source [11]).

`train` This network contains contacts between suspected terrorists involved in the train bombing of Madrid on March 11, 2004 as reconstructed from newspapers. ($n = 62$, $m = 243$, source [6]).

`email` This is the email communication network at the University Rovira i Virgili in Tarragona in the south of Catalonia in Spain. ($n = 1133$, $m = 5451$, source [5]).

`powergrid` This network is the high-voltage power grid in the Western States of the United States of America. ($n = 4941$, $m = 6594$, source [12]).

				Deccen		Multi-BFS	
Network	n	m	Δ	Steps	Messages	Steps	Messages
`surf`	43	336	3	6	181374	7	28896
`dolphins`	62	159	8	16	140051	17	19716
`macaques`	62	1167	2	4	1458368	5	144708
`train`	64	243	6	12	272673	13	31104

Table 1: Steps required to complete and number of exchanged messages by Deccen and Multi-BFS executed with $p = 1$.

5.1 Performance comparison

The performance measures used to evaluate the algorithms are the number of steps required to complete the computation and the number of messages that the network nodes generate.

Table 1 reports the results obtained by running Deccen and Multi-BFS with $p = 1$ in order to compute the exact centrality values. Due to the high memory requirements of Deccen the simulations could only be performed on small networks, but both algorithms can be expected to perform in a similar way (relatively to each other) in any other scenario since all the messages are generated in a deterministic way.

As expected, in both cases the number of steps required to complete is twice the diameter Δ of the network, since each "discovery" phase takes at most Δ steps to reach any destination from a given source, and the same also holds for the report phase of Deccen or the backtracking in Multi-BFS. In all cases, Multi-BFS requires an extra step between reaching the farthest child nodes and the beginning of the backtracking phase: this is bound to happen whenever nodes at distance Δ from any source have siblings, since they need to be detected before reporting back.

The experimental data shows that Deccen requires a significantly larger amount of messages than Multi-BFS to terminate the execution, as anticipated by the theoretical analysis of the algorithms. This is a consequence of the different way in which the algorithms handle the report phase: while in Deccen new reports are broadcast at every step of a discovery, Multi-BFS uses the distance from the source of the discovery to rank the nodes, and only generates reports from the most distant nodes (that are then routed back to the source using the predecessor sets).

5.2 Multi-BFS approximations

The quality of the approximations obtained with Multi-BFS were evaluated by running simulations with increasing values of p (from 0.05 to 1.0 with an increment of 0.05 at each step); each experiment was repeated 10 times. The aim of the experiments was to measure the numerical error of the estimates yielded by the algorithm, and to evaluate how close the "ranking" of the nodes using these estimates is to the actual ranking of the nodes according to the exact centrality values, since nodes could be ranked correctly even if the estimates are subject to high numerical error. This second experiment is especially interesting if a node wants to make a "greedy" choice knowing only its and it's neighbors index's value as it gives an idea of how often this choice may be the wrong one.

Figure 1 reports the average relative error ϵ_r yielded by the centrality estimators in the `dolphins`, `email` and `powergrid` networks. The results show satisfactory results for the estimation of closeness centrality for small values of p, while stress and betweenness estimates exhibit large numerical error even for values of p close to 1.

The accuracy of the ranking induced by the approximated indices is measured by counting, among all the pairs of nodes, the fraction of pairs in which the nodes are wrongly ordered with respect to the ranking imposed by the exact centrality values; results are reported in figure 2. In this case the results are encouraging: even for small values of p the fraction of pairs in wrong rank order is relatively small.

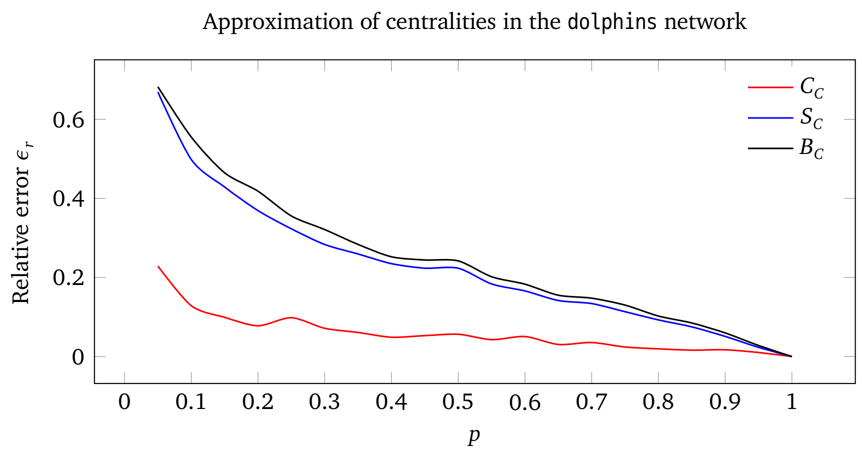
6 Project code overview

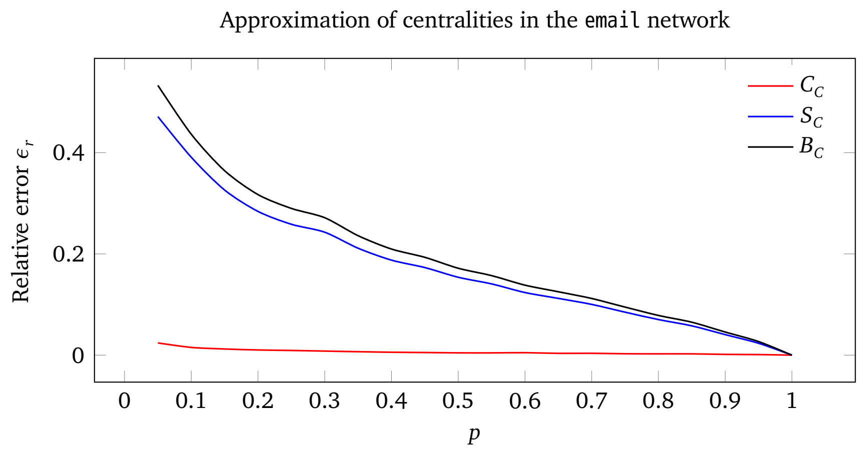
The simulations were designed to work with the *cycle-driven* engine of the PeerSim [10] simulator. This section explains how the protocol classes were adapted to comply to the synchronous communication model, and then illustrates the main features of the implemented protocols.

6.1 Synchronous communication model

Implementing the algorithms as cycle-driven protocols seems a natural choice given the assumption of synchronous communication under which the algorithms are specified: the cycle-driven engine simply iterates over each `Node` in the network array and executes the protocol logic at each step. However, to correctly model the synchronous communication, a protocol shouldn't be able to influence another one in the middle of a cycle, since if this happens while the second protocol is yet to be executed, it would receive information one cycle (or step) in advance. In other words, each protocol must be executed in isolation from the others at each cycle, and a message generated during a cycle should reach the destination only at the beginning of the following one.

To solve this issue, whenever a new message is generated it must be retained by the sender protocol until the iteration of the current cycle is finished, and then delivered to the receiver protocol. This can be achieved in a number of different ways; the choice made here was to define a generic interface providing methods to allow *synchronous* protocols to communicate with each other under





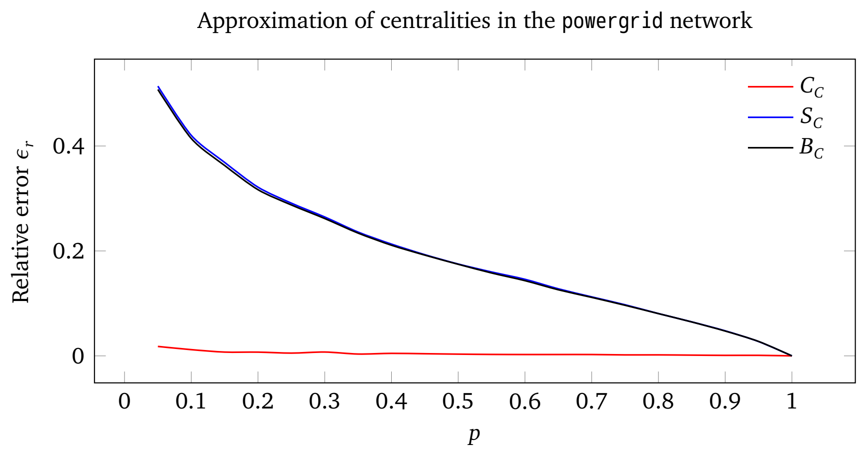


Figure 1: Approximation error in the estimation of centrality indices.

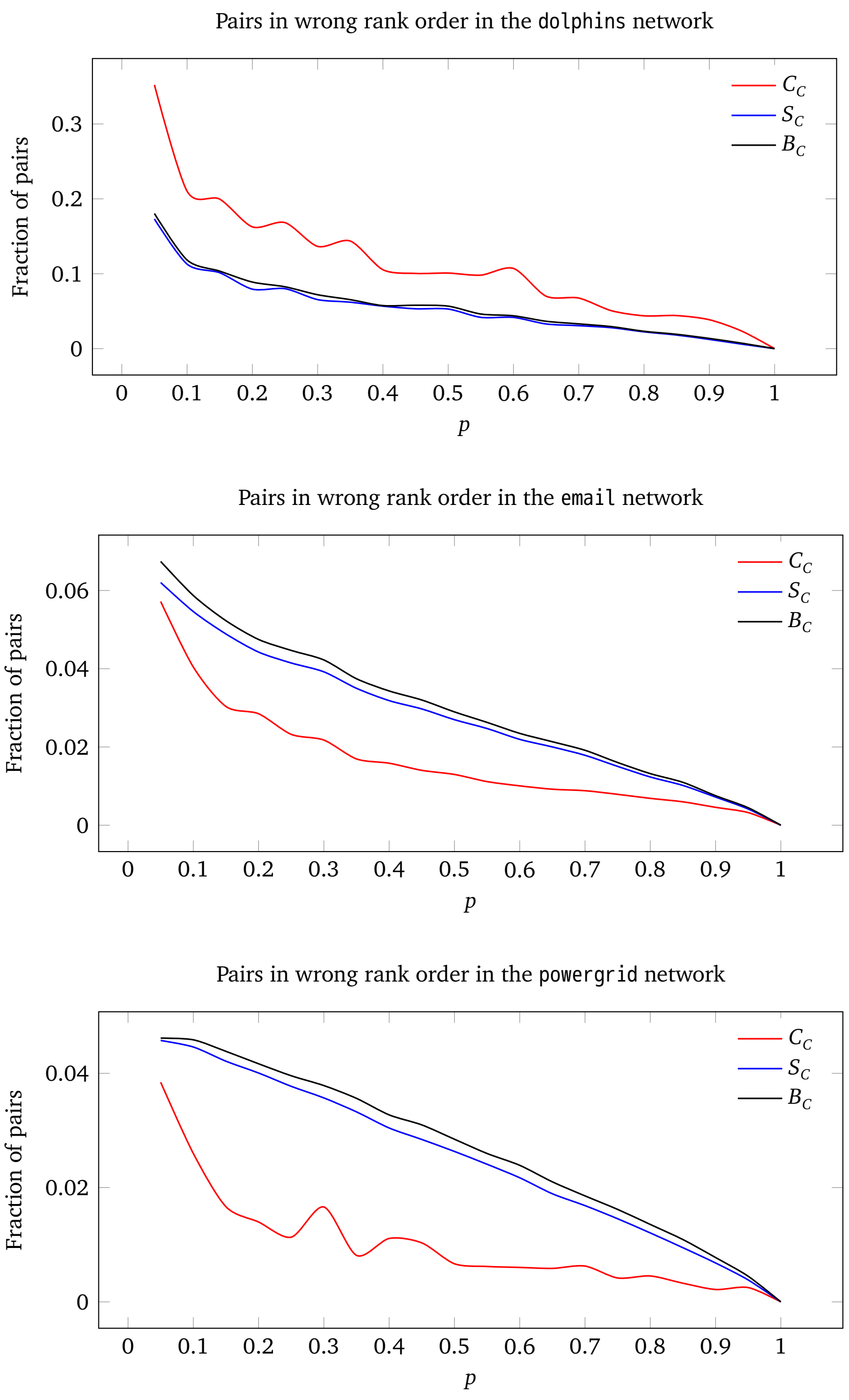


Figure 2: Pairs in wrong rank order with approximated centralities

the requirement above but in a way that is transparent to the logic of the protocol itself, and defining a PeerSim `Control` to move deliver messages at the end of each cycle.

6.1.1 Components

The `CycleBasedTransportSupport` interface is the generic interface that synchronous protocol classes are required to implement (its type argument is the class that models the messages protocols exchange with each other):

```
public interface CycleBasedTransportSupport<T> {
  interface SendQueueEntry<T> {
      T getMessage();
      CycleBasedTransportSupport<T> getDestination();
  }
  void addToSendQueue(
      T message, CycleBasedTransportSupport<T> destination);
  boolean hasOutgoingMessages();
  void addToIncoming(T message);
  Iterator<T> getIncomingMessagesIterator();
  Iterator<SendQueueEntry<T>> getSendQueueIterator();
}
```

A protocol invokes the `addToSendQueue` method to "send" a message to the specified `destination` argument. The message is not actually sent, but held in a container until the cycle ends and the `CycleBasedTransport` control begins the execution. The control then obtains an iterator to the send queue of each protocol (with the `getSendQueueIterator` method), and moves the messages to the destinations by calling the `addToIncoming` method on them with the messages as arguments. A protocol can access available messages by obtaining an `Iterator` over them with the `getIncomingMessagesIterator` method.

6.2 Protocol classes

Protocol classes are organized as shown in figure 3. Both `Deccen` and `Multi-BFS` extend the abstract class `SynchronousCentralityProtocol`, which provides an implementation the `CycleBasedTransportSupport<Message>` interface and declares the `nextCycle` method of the `CDProtocol` PeerSim interface `abstract`.

6.2.1 Messages

The messages exchanged by the protocols during the simulation are instances of the `Message` class. This class provides factory methods to generate DISCOVERY and REPORT message objects that follow the conventions used in the definition of the algorithms.

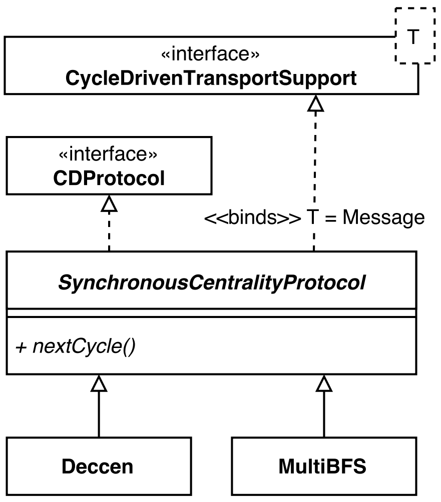


Figure 3: Protocol classes.

6.2.2 DECCEN protocol implementation

The `Deccen` class implements a DECCEN node in the simulator. A node state consists of the shortest path information it collects and of the reports it has handled during the execution:

```
public class Deccen extends SynchronousCentralityProtocol {
  private static class ShortestPathData {
    public final int count;
    public final int length;

    public ShortestPathData(int count, int length) { ... }
  }

  private static class OrderedPair<T1,T2> {
    public final T1 first;
    public final T2 second;

    public OrderedPair(T1 first, T2 second) { ... }
    ...
  }
  ...
  private Map<Node,ShortestPathData> shortestPathMap;
  private Set<OrderedPair<Long,Long>> handledReports;
  ...
  public void nextCycle(Node self, int protocolID) { ... }
  ...
}
```

Whenever new DISCOVERY messages are received, additional information about the number and length of shortest paths toward a particular source becomes available: this information is stored in the `shortestPathMap` structure by adding a new `ShortestPathData` entry paired with the source of the discovery. Since the algorithm computes all the centrality indices, it stores both the distance from the source and the number of different paths counted. Reports about a source–destination pair (s, t) are tracked by storing the `handledReports` set an `OrderedPair` of `Node` IDs (which are `long` integers in PeerSim).

The implementation of `nextCycle` simply executes the actions described in section 3.4:

```
public void nextCycle(Node self, int protocolID) {
  Map<Node,List<Message>> discoveryMap =
      new HashMap<Node,List<Message>>();
  List<Message> reportList = new LinkedList<Message>();
  parseIncomingMessages(discoveryMap, reportList);
  if (!discoveryMap.isEmpty())
    processDiscoveryMessages(self, protocolID, discoveryMap);
  if (!reportList.isEmpty())
    processReportMessages(self, protocolID, reportList);
}
```

6.2.3 Multi-BFS protocol implementation

The state of a Multi-BFS node consists of a `Map` to keep track of the various visits that are performed by the source nodes during the algorithm execution:

```
public class MultiBFS extends SynchronousCentralityProtocol {
  private static class VisitState {
    ...
  }
  ...
  private Map<Node, VisitState> activeVisits;
  private Set<Node> completed;
  ...
  public void nextCycle(Node self, int protocolID) { ... }
  ...
  public boolean isWaiting(Node source) { ... }
  public boolean isActive(Node source) { ... }
  public boolean isCompleted(Node source) { ... }
}
```

Recall that the state of a node is parametric with respect to each source of a visit. The state is WAITING(`s`) if the `Node` instance `s` does not appear as key in the `activeVisits` map and neither is contained in the `completed` set; this is the initial state for any source since both structures are empty at the start of the protocol. When a node is in state ACTIVE(s) an entry with key s is present in the activeVisits map. After a node has reported back to all the predecessors, it changes state COMPLETED(s) by removing the mapping from activeVisits and inserting s in the completed set.

Entries in the activeVisits map are used to store data while a node is in the “active” phase. A VisitState object is used to keep track of the predecessors and children sets, and to incrementally compute the values to be reported to the predecessor nodes:

```
private static class VisitState {
  public Set<Node> predecessors;
  public Set<Node> siblings;
  public Set<Node> children;
  public int distanceFromSource;
  public int timestamp;
  public int sigma;
  public long contributionSC;
  public double contributionBC;
  ...
  public void accumulate(
      Node child, double bcc, long scc, int numSP) { ... }
}
```

The accumulate method updates the local contributions of the source by applying the recursive relations (8) and (9), and it’s invoked whenever a child node sends a report.

Finally, the nextCycle method follows the specification given in section 4.4.5:

```
public void nextCycle(Node self, int protocolID) {
  Map<Node,List<Message>> discoveryMap =
      new HashMap<Node,List<Message>>();
  List<Message> reportList = new LinkedList<Message>();
  parseIncomingMessages(discoveryMap, reportList);
  if (!discoveryMap.isEmpty())
    processDiscoveryMessages(self, protocolID, discoveryMap);
  if (!reportList.isEmpty())
    processReportMessages(reportList);
  reportNewCompleted(self, protocolID);
}
```

Messages are parsed and handled by type, then active visits for which all child nodes have reported are finalized with the reportNewCompleted method by integrating the contributions in the centrality accumulators and reporting the relevant data to each predecessor.

Appendix: Theorem proofs

Theorem 2. *The stress centrality contribution of $s \in V$ on any $v \in V$ obeys*

$$\sigma(s|v) = \sum_{w:v \in P_s(w)} \sigma_{sv} \cdot \left(1 + \frac{\sigma(s|w)}{\sigma_{sw}}\right). \tag{9}$$

Proof. The proof is analogous to the one provided by Brandes in [1] to prove Theorem 1.

Let $\sigma_{st}(v, e)$ be the number of shortest paths between s and t that pass through v and across edge e. Observe that if a shortest path from s to t passes through v, then after v it must immediately reach some other node w that has v in its predecessor set $P_s(w)$, so equation (6) can be rewritten as

$$\begin{aligned}\sigma(s|v) &= \sum_{t \in V} \sigma_{st}(v) = \sum_{t \in V} \sum_{w:v \in P_s(w)} \sigma_{st}(v, \{v, w\}) \\ &= \sum_{w:v \in P_s(w)} \sum_{t \in V} \sigma_{st}(v, \{v, w\}).\end{aligned}$$

Let w be any node with $v \in P_s(w)$, then

$$\sigma_{st}(v, \{v, w\}) = \begin{cases} \sigma_{sv} & \text{if } t = w \\ \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sigma_{st}(w) & \text{if } t \neq w \end{cases}$$

and substituting it in the previous expression yields

$$\begin{aligned}\sigma(s|v) &= \sum_{w:v \in P_s(w)} \sum_{t \in V} \sigma_{st}(v, \{v, w\}) \\ &= \sum_{w:v \in P_s(w)} \left(\sigma_{sv} + \sum_{t \neq w} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sigma_{st}(w)\right) \\ &= \sum_{w:v \in P_s(w)} \sigma_{sv} \cdot \left(1 + \frac{\sigma(s|w)}{\sigma_{sw}}\right).\end{aligned}$$

□

Theorem 3 (Unbiased estimators)**.** *The expected values of the centrality estimators are the actual centrality indices:*

(a) $\mathbf{E}[\widetilde{C}_C(u)] = C_C(u)$,

(b) $\mathbf{E}[\widetilde{C}_S(u)] = C_S(u)$,

(c) $\mathbf{E}[\widetilde{C}_B(u)] = C_B(u)$.

Proof. (a) Recall that for the estimation of closeness centrality, the result of sampling from a source v_i at a node u is modeled with the random variable $X_i(u) = \frac{n}{n-1} \cdot d(v_i, u)$. The derivation exploits the linearity of the expected value operator **E** and the fact that source nodes are random (that is, each node has equal probability $1/n$ of being a source).

$$\begin{aligned}
\mathbf{E}[\widetilde{C}_C(u)] = \mathbf{E}\left[\sum_{i=1}^{k}\frac{X_i(u)}{k}\right] &= \mathbf{E}\left[\sum_{i=1}^{k}\frac{n\cdot d(v_i,u)}{k(n-1)}\right] \\
&= \frac{n}{k(n-1)}\sum_{i=1}^{k}\mathbf{E}\left[d(v_i,u)\right] && \text{(by linearity of expectation)} \\
&= \frac{n}{k(n-1)}\cdot k\cdot\frac{1}{n}\sum_{v\in V} d(v,u) && \text{(random source selection)} \\
&= \frac{\sum_{v\in V} d(v,u)}{n-1} = C_C(u)
\end{aligned}$$

(b) To estimate stress centrality, the result of sampling from v_i is modeled at u with the random variable $Y_i(u) = n\cdot\sigma(v_i|u)$. The derivation relies again on the linearity of **E** and the uniform distribution of source nodes.

$$\begin{aligned}
\mathbf{E}[\widetilde{C}_S(u)] = \mathbf{E}\left[\sum_{i=1}^{k}\frac{Y_i(u)}{k}\right] &= \mathbf{E}\left[\sum_{i=1}^{k}\frac{n\cdot\sigma(v_i|u)}{k}\right] \\
&= \frac{n}{k}\sum_{i=1}^{k}\mathbf{E}\left[\sigma(v_i|u)\right] = \frac{n}{k}\cdot k\cdot\frac{1}{n}\sum_{v\in V}\sigma(v|u) = C_S(u)
\end{aligned}$$

(c) The proof is the same as (b) with the substitution of $Z_i(u) = n\cdot\delta(v_i|u)$ for $Y_i(u)$. □

References

[1] Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, **25**(2), 163–177.

[2] Brandes, U. and Pich, C. (2007). Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, **17**(07), 2303–2318.

[3] Eppstein, D. and Wang, J. (2004). Fast approximation of centrality. *J. Graph Algorithms Appl.*, **8**(1), 39–45.

[4] Freeman, L. C., Freeman, S. C., and Michaelson, A. G. (1988). On human social intelligence. *J. of Social and Biological Structures*, **11**(4), 415–425.

[5] Guimerà, R., Danon, L., Díaz-Guilera, A., Giralt, F., and Arenas, A. (2003). Self-similar community structure in a network of human interactions. *Phys. Rev. E*, **68**(6), 065103.

[6] Hayes, B. (2006). Connecting the dots. can the tools of graph theory and social-network studies unravel the next big plot? *American Scientist*, **94**(5), 400–404.

[7] Kunegis, J. (2013). KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350.

[8] Lehmann, K. A. and Kaufmann, M. (2003). Decentralized algorithms for evaluating centrality in complex networks.

[9] Lusseau, D., Schneider, K., Boisseau, O. J., Haase, P., Slooten, E., and Dawson, S. M. (2003). The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, **54**, 396–405.

[10] Montresor, A. and Jelasity, M. (2009). PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA.

[11] Takahata, Y. (1991). Diachronic changes in the dominance relations of adult female Japanese monkeys of the Arashiyama B group. *The Monkeys of Arashiyama. State University of New York Press, Albany*, pages 123–139.

[12] Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, **393**(1), 440–442.