# Peer-to-Peer Systems

Andrea Maggiordomo – mggndr89[at]gmail.com
Laurea Magistrale in Informatica, Università di Pisa

2016

## 1 Introduction

The task of the project was to define DECCEN, a decentralized algorithm able to compute the stress centrality values of a network (based on the ideas introduced in [9]), and study its behavior using the *Peersim* simulator [10]. 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 computation could be extended to also include closeness and betweenness centrality values.

This report defines DECCEN extended to compute the three centrality indices mentioned. Furthermore, a different decentralized algorithm is defined, based on the previous works of Brandes, and Eppstein and Wang [2, 4, 3], to approximate centrality indices by involving only a limited amount of agents in the computation. This second algorithm, labeled MULTI-BFS, has a significantly smaller communication overhead than DECCEN.

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: 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. Follow a small overview of the most relevant choices made in the development of the project code, and the proofs of some results introduced in section 4.

## 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. As usual, $n$ is the number of nodes $|V|$ and $m$ is the number of edges $|E|$. Each node $v \in V$ represents an independent agent with some given computational power, that can only communicate with its neighbors $N_v = \{u \in V : \{u, v\} \in E\}$. A path $p(s, t)$ from a source $s$ to a destination $t$ is a sequence of edges that connects the two endpoints. The distance $d(u, v)$ between two nodes is the length of the

shortest path that connects them (with $d(u,u) = 0$) while the diameter $\Delta$ is the maximum distance between any pair of nodes. Note that $d(u,v) = d(v,u)$ since the graph is undirected. A node $v$ is a predecessor of $w$ with respect to a source $s$ if $\{v,w\} \in E$ and $d(s,v)+1 = d(s,w)$. The *predecessor set* $P_s(w)$ of $w$ is the set of all predecessors of $w$ with respect to $s$.

The number of different shortest paths that connect two nodes $s,t \in V$ is denoted by $\sigma_{st}$, 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 node $v \in V$ is defined as

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

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

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

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

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

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

# 3 The Deccen algorithm

The specification of Deccen is based on the algorithmic scheme outlined in [9]. 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 $\sigma_{st}$ 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** (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 $\sigma_{vs}$ and $\sigma_{vt}$. 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 $\sigma_{sv}$ 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 $(s, \sigma_{sv})$ 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 $n^2m$.

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 issue with the Deccen algorithm is its computational cost both in terms of memory consumption and number of messages exchanged, rendering the algorithm impractical for networks of reasonable size.. In order to keep track of the forwarded report messages and guarantee the termination of the protocol each node needs to maintain a data structure of size $O(n^2)$, while the number of messages exchanged is $O(n^2m)$.

However, if we are for example interested in computing centrality indices in order to mitigate network congestion a simple estimation of the values could be sufficient. In the following I propose an approximation algorithm that adapts an idea originally developed for closeness centrality in [4] and expanded for the estimation of betweenness centrality in [3]. The idea is to isolate the contribution of a single node $s$ to the centrality values of all the other nodes in the network and compute those contributions by solving a Single-Source-Shortest-Path problem starting from $s$. We can then compute the centrality of a node $v$ by adding the contributions of all the other nodes to $v$ (that is, by solving $n$ SSSP instances starting from all the nodes and accumulating the contributions locally). In our case, the SSSP problem is converted to a decentralized Breadth First Search.

The approximation algorithms described in [4, 3] estimate the centrality values by solving the SSSP problems for a restricted set of source nodes.

### Contribution of a source to Closeness centrality

The contribution of a source $s$ to the closeness centrality value of $v$ is simply the distance $d(s, v)$ of $v$ from $s$:

$$\gamma(s|v) = d(s, v). \tag{4}$$

### 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 [2]:

$$\delta(s|v) = \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{5}$$

that allows to rewrite the betweenness centrality of $v$ as

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

### Contribution of a source to Stress centrality

The contribution to stress centrality is analogous to the betweenness centrality:

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

and the stress centrality of $v$ is rewritten as

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

## 4.1 Computing contributions from a single source

As stated previously, 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 (that is, the depth of the visited node in the Breadth-First tree) and it can be computed directly during the visit.

For betweenness centrality, [2] shows 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*

$$\delta(s|v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta(s|w)). \tag{7}$$

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*

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

The predecessor sets of all the nodes can be easily discovered by adjusting the "descent" phase of the BFS algorithm, while contributions are computed during a backward walk from the frontier of the BF-Tree back to the source.

### 4.2 Random sampling of source nodes [FIXME]

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 of samples is not known beforehand, eventually all the nodes will become aware of the number of sources that initiated a visit (let it be $k$).

The contributions of sample $v_i$ to the closeness, stress and betweenness centrality of any node $u$ can be modeled with the following random variables

To estimate the value of centrality indices, first the result of sampling from a source node $v_i$ at a node $u$ is modeled using the following random variables:

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

then centrality indices are approximated with the following estimators:

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

The derivations for the following 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 exact centrality indices:*

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

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

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

## 4.3 Algorithm specification

In this section the algorithm is detailed. The only parameter of the algorithm is the probability $p$ with which each independent node decides to begin a visit of the network. The network size $n$ is assumed to be known to all the agents in the network. Initially, every node sets to zero the estimation of each centrality index.

### 4.3.1 Message types

**DISCOVERY**$\langle s, u, d, \sigma_{su} \rangle$ Messages of this type are used during the descent from a source to build the predecessors sets at each node. Relevant 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 $\sigma_{su}$.

**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 backtracking phase to inform its predecessors of the computed contributions and to allow them to compute their own by applying the recursive relations introduced in section 4.1.

### 4.3.2 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, updated them accordingly, and reported the contributions to each predecessor in $P_s(v)$ by sending the appropriate REPORT messages.

### 4.3.3 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 protocol.

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 $C_S(s)$.

### 4.3.4 Protocol initialization

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

### 4.3.5 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 $\mathsf{DISCOVERY}\langle s, u, d, \sigma_{su}\rangle$ messages having source $s$ and distance $d$ with $v$ in state $\mathsf{WAITING}(s)$:

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

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

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

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

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

   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 $C_B(s) \leftarrow C_B(s) + \sigma_{sw} \cdot (1 + \sigma(s|w)/\sigma_{sw})$.

5. For any $s \in V$ such that $v$ is in state $\mathsf{ACTIVE}(s)$, 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 $\mathsf{REPORT}\langle s, v, \delta(s|v), \sigma(s|v), \sigma_{sv}\rangle$ message to all the predecessors contained in $P_s(v)$.

5.2. Increment the counter $k$ and change state to $\mathsf{COMPLETED}(s)$.

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

$$\widehat{C}_C(v) = \left(\frac{n}{k}\, C_C\right)^{-1}, \quad \widehat{C}_B(v) = \frac{n}{k}\, C_B, \quad \widehat{C}_S(v) = \frac{n}{k}\, C_S.$$

Note that if the algorithm is executed with $p = 1$ this is basically the decentralized version of the algorithm described in [2], and the estimators are actually the exact centrality indices.[FIXME move this at the beginning of the description]

### 4.4 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 $\mathsf{ACTIVE}(s)$ 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 $O(\lceil pn \rceil deg(v))$.

## 5 Experiments

Simulations were performed to evaluate the performances of the algorithms, and to assess the quality of the estimations obtained by running Multi-BFS with different values of the parameter $p$.

The networks used in the experiments (taken from the KONECT repository [8]) are:

**`dolphins`** This is a social network of bottlenose dolphins. The nodes are the bottlenose dolphins (genus *Tursiops*) of a bottlenose dolphin community living off Doubtful Sound, a fjord in New Zealand. An edge indicates a frequent association. The dolphins were observed between 1994 and 2001. ($n = 62$, $m = 159$, source [1]).

**`surf`** This network contains interpersonal contacts between windsurfers in southern California during the fall of 1986. ($n = 62$, $m = 336$, source [5]).

| | | | | Deccen | | Multi-BFS | |
|---|---|---|---|---|---|---|---|
| Network | $n$ | $m$ | $\Delta$ | Steps | Messages | Steps | Messages |
| `surf` | 43 | 336 | 3 | 6 | 1160310 | 6 | 28896 |
| `dolphins` | 62 | 159 | 8 | 16 | 985403 | 16 | 19716 |
| `macaques` | 62 | 1167 | 2 | 4 | 8708620 | 4 | 144708 |
| `train` | 64 | 243 | 6 | 12 | 1729397 | 12 | 31104 |

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

`macaques` This directed network contains dominance behaviour in a colony of 62 adult female Japanese macaques. An undirected version of the network (where edges are made symmetric) was used to run the experiments. ($n = 62$, $m = 1167$, 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 [7]).

`email` This is the email communication network at the University Rovira i Virgili in Tarragona in the south of Catalonia in Spain. Edges represent contacts between users. ($n = 1133$, $m = 5451$, source [6]).

`powergrid` This network is the high-voltage power grid in the Western States of the United States of America. The nodes are transformers, substations, and generators, and the ties are high-voltage transmission lines. ($n = 4941$, $m = 6594$, source [12]).

### 5.1 Performance analysis

The performance metrics used to evaluate the algorithms were the number of steps required to complete the computation and the number of messages that the agents needed to generate to do so. Note that due to the high memory requirements of Deccen, these simulations were performed on small networks.

Table 1 reports the results obtained by running Deccen and Multi-BFS with $p = 1$ in order to compute the exact centrality values. As expected, in both cases the number of steps required to complete is exactly twice the diameter $\Delta$ of the networks, since each "discovery" phase takes at most $\Delta$ 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. However, the number of messages required by Deccen is significantly larger. This follows from the different way in which the report phase evolves in the two algorithms: while in Deccen any node must generate a report for each

source and broadcast it to every other node in the network, in MULTI-BFS reports are routed back to the source by signaling only the predecessors at each step.

### 5.2 Multi-BFS approximations

The quality of the estimations yielded by MULTI-BFS was evaluated both in terms of the numerical error introduced by the estimators, and the difference in the ranking of the nodes induced by the centrality values which could prove to be accurate even in presence of non-negligible error in the estimates.

Figure 1 reports the average relative error $\epsilon_r$ yielded by the centrality estimators for increasing values of the parameter $p$ in the `dolphins`, `email` and `powergrid` networks. The estimation of Closeness centrality is much more accurate than the estimation of Stress and Betweenness centrality, which behave roughly the same.

The accuracy of the ranking is measured by counting among all the pair of nodes, the fraction of pairs in which the nodes are wrongly ordered with respect to the ranking induced by the exact centrality values. Results are reported in figures 2–4. In this case, even for small values of $p$ the fraction of pairs wrongly ranked is relatively small. This is encouraging if the indices are used locally, to make decisions based on the values computed at a node and at its neighbors.

## Appendix A: theorem proofs

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

$$\sigma(s|v) = \sum_{w:v\in P_s(w)} \sigma_{sv} \cdot \left(1 + \frac{\sigma(s|w)}{\sigma_{sw}}\right). \tag{5}$$

*Proof.* The proof is analogous to the one provided by Brandes in [2] 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 $t$ travels 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

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

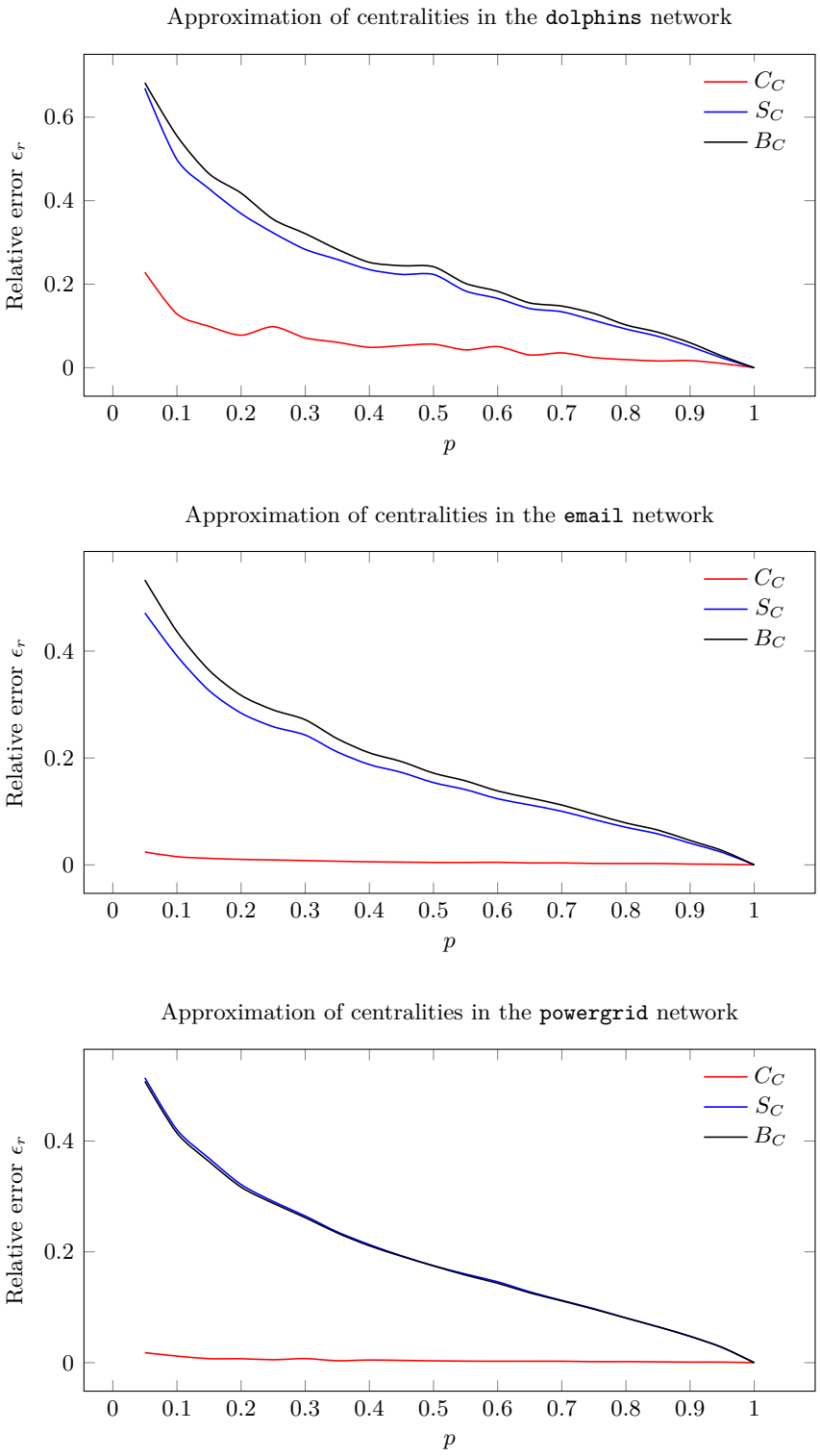


Figure 1: Approximation error in the estimation of centrality indices.

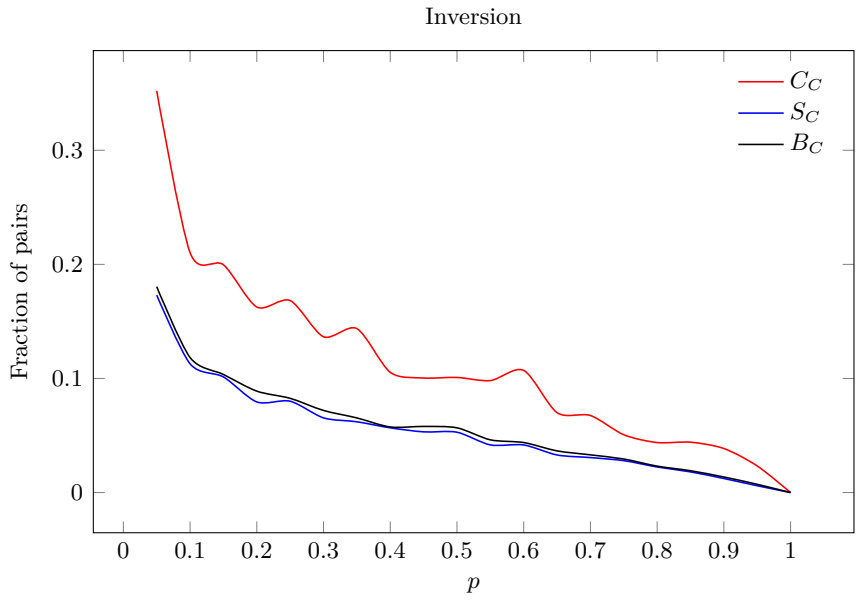


Figure 2: Fraction of pairs in wrong rank order in the `dolphins` network

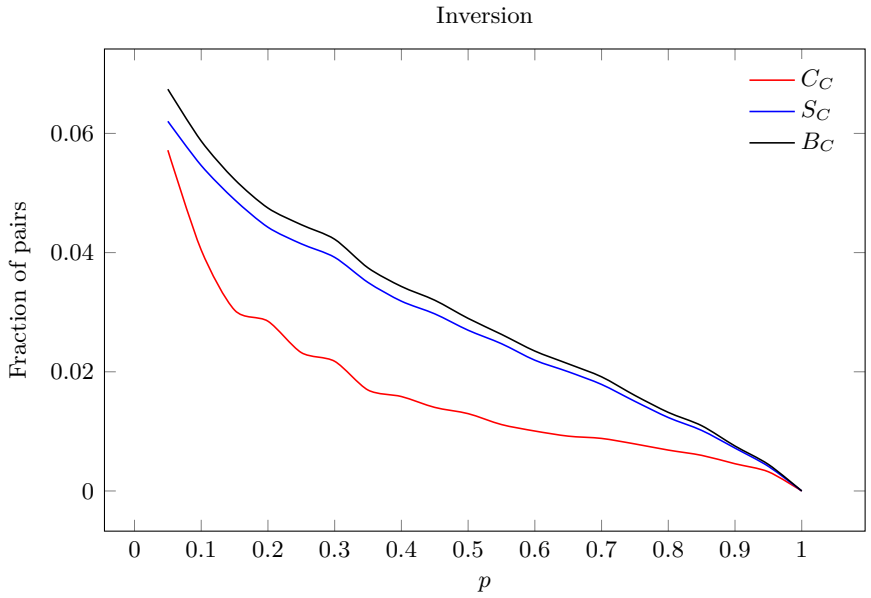


Figure 3: Fraction of pairs in wrong rank order in the `arenas-email` network

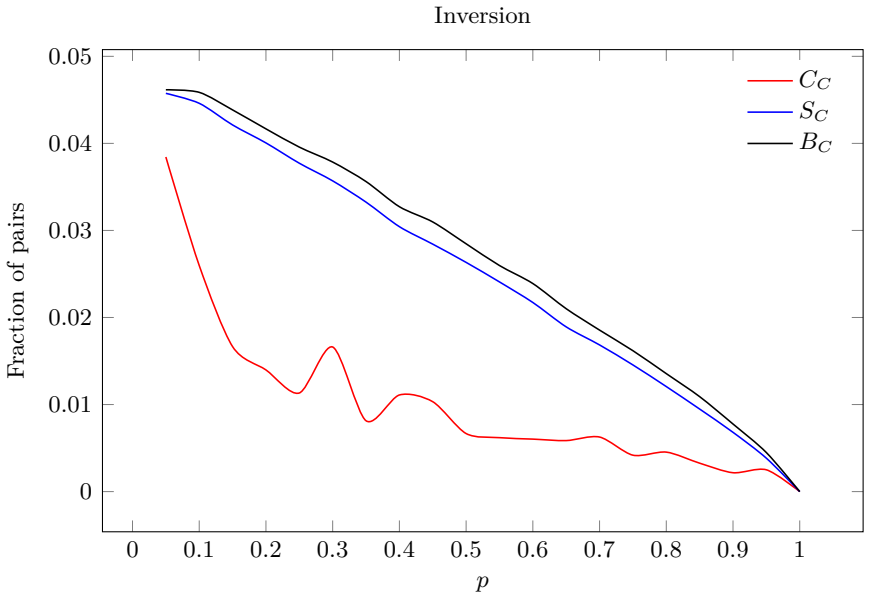


Figure 4: Fraction of pairs in wrong rank order in the `opsahl-powergrid` network

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 exact centrality indices:*

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

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

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

*Proof.* (a) Recall that for the estimation of closeness centrality, the result of sampling from a source $v_i$ at a node $u$ was 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 $\mathbf{E}$ and the fact that source nodes are random (that is, each node has equal probability $1/n$ of being a source).

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

(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 $\mathbf{E}$ and the uniform distribution of source nodes.

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

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

## Appendix B: project code overview

Simulations are based on the *cycle-driven* engine of PeerSim. This seems a natural choice given the synchronous communication model assumption, but the way in which the engine handles the execution of the protocol code required some adjustments. When dealing with cycle-driven protocols, PeerSim executes each simulation cycle sequentially by iterating through the `Node` instances that form the network and invoking the `nextCycle` method that the protocol implements. If precautions are not taken, this execution policy clashes with the synchronous model assumption: if a node $n$ has two neighbors that at the current step need to interact with it (for example to communicate the number of shortest paths) and one of them is executed after $n$ in the iteration order, the information will be available to $n$ only at the following cycle.

Rather than adapting the protocol code to be mindful of this potential situation, the choice was to include the synchronous communication model

in the simulation. A synchronous protocol is a `CDProtocol` that can send messages to other synchronous protocols with the guarantee that those will be received after the `nextCycle` method will be invoked by the simulator on the receiver at the current step, but before the invocation at the following step. This allows to execute each simulation cycle at each node in isolation, only using information generated in the previous cycle exactly as the synchronous communication model assumption requires. This is achieved by implementing the `CycleBasedTransportSupport` interface:

```
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();
}
```

This generic interface offers facilities to synchronously transfer objects between protocols that implement it. The `addToSendQueue` method should temporarily store the message and the destination in a send queue, which is then iterated by a `CycleBasedTransport` control at the end of a cycle in order to deliver each message to the appropriate destination. A protocol can then call the `getIncomingMessageIterator` method in the next cycle and iterate through the received messages.

### 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.

### Protocol classes

Protocol classes are organized in the hierarchy shown in figure 5. The abstract class `SynchronousCentralityProtocol` implements the `CycleBasedTransportSupport<Message>` interface, while the `nextCycle()` method of the PeerSim interface `CDProtocol` is declared `abstract`.

The implementation of the protocols is straightforward and very similar to the descriptions given previously.

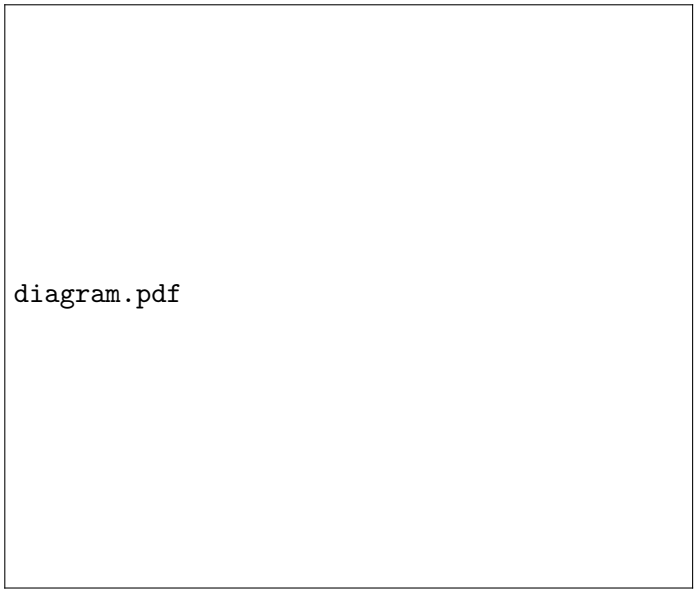


Figure 5: Class diagram [TODO]

### Deccen protocol implementation

The `Deccen` class implements a DECCEN node in the simulator. A node needs to keep track of the shortest path information it has collected and of the reports it has handled.

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

        public ShortestPathData(int count, int length) {
            this.count = count;
            this.length = length;
        }
    }

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

        public OrderedPair(T1 first, T2 second) {
            this.first = first;
            this.second = second;
        }
```

```
        ...
    }
    ...
    private Map<Node, ShortestPathData> shortestPathMap;
    private Set<OrderedPair<Long,Long>> handledReports;
    ...
}
```

Listing ?? reports an excerpt of the `Deccen` class. The `shortestPathMap` map is used to store relevant information about the shortest paths to other nodes and is updated whenever new DISCOVERY messages are received while reports are tracked with `handledReports` set of `Node` ID pairs.

The `nextCycle()` method 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, discover
    if (!reportList.isEmpty()) processReportMessages(self, protocolID, reportList);
}
```

### Multi-BFS protocol implementation

The state of a MULTI-BFS node mainly consists of a dictionary to keep track of the various visits that are performed during the algorithm execution.

```
public class MultiBFS extends SynchronousCentralityProtocol {
    private static class VisitState {
        ...
    }
    ...
    private Map<Node, VisitState> activeVisits;
    private Set<Node> completed;
    ...
    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 centrality contributions by applying the recursive relations (7) and (8), and is invoked whenever a child node sends a report.

Finally, the implementation of the nextCycle() method which closely follows the specification given in section 4.3.5:

```
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, discover
    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.

## References


[1] (2003). The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, **54**, 396–405.

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

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

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

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

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

[7] 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.

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

[9] Lehmann K. A.; Kaufmann M. (2003). Decentralized algorithms for evaluating centrality in complex networks.

[10] Montresor A.; Jelasity M. (2009). PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pp. 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*, pp. 123–139.

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