A Self-stabilizing Spanning Tree Approach to Distributed Averaging

Anvesh Tanuku

Advisor: Randy Freeman

EECS 399

December, 2012

Contents

1 Abstract

In this paper, we have designed a robust method for distributed average consensus in a network where topology is dynamically changing for some finite time. The method involves creating a self stabilizing spanning tree on the network and using it to quickly compute the global signal process. Our algorithm works for both sinusoidal signals with unknown amplitude and phase as well as constant signals. We test the new method's response to network faults and empirically demonstrate the comparative behavior of this method to the standard PI estimator approach.

2 Introduction

We present a method of distributed averaging in a dynamically changing network. In this project, we have a network with a dynamic topology: nodes may enter and leave the network and communication links may change in time. Each node in the network has a local reading of an input signal that may or may not be time varying. The goal is to have each node asymptotically track the average of all such signals in the network. A previous approach to this problem was proposed by (Bai et al.,2010). In discrete time:

$$\dot{v}_i[k+1] - v_i[k] = \gamma(\phi_i - v_i) - K_p \sum_{j \in N_i} a_{ij}(v_i - v_j) - K_I \sum_{j \in N_i} a_{ij}(\nu_i, \nu_j) \tag{1}$$

$$\dot{\nu}_i[k+1] - \nu_i[k] = K_I \sum_{j \in N_i} a_{ij}(v_i, v_j) \tag{2}$$

Here, N_i denotes the set of all neighbors of the ith agent, $\phi_i \in \mathbb{R}$ is agent i's input to the estimator. $v_i \in mathbbR$ is the estimate of the ith agent of the network, ν_i is the internal state of agent i. The terms a_{ij} are weights for each edge in the graph. Regardless of the values of initial condition, this model has been proven to converge to the global average of all signals in a constant input graph for appropriate choices of the gains. In discrete time, acceptable choice of the network bounds relies in part on the upper bound of the network. Since we are looking for a decentralized appraoch we assume each agent has no such access to an upper bound. Here we propose a separate algorithm: an algorithm that maintains a self stabilizing spanning tree on the network. This new algorithm requires no knowledge of the number of nodes in the network and is robust to network faults. We detail some more of the model assumptions and design below.

3 Design Constraints and Requirements

In this section we detail the assumptions and constraints in the development of the spanning tree algorithm. We assume our network is dynamic. That is, links can come into operation and drop out of operation during a transient period. Furthermore, the neighbor lists of the nodes may also change in time. Finally we make the assumption that packet losses can occur during the course of running our algorithm. Nodes in the network read their neighbor's information through communication channels. So we must allow either partial or complete failure of We assume that the frequency of these perturbations is lower than the time to convergence of our algorithm. This amounts to certain periods of time where no perturbations take place. Computations are atomistic– that is read/write operations that require information from other nodes occur instantaneously. Node computations are carried out by a fair scheduler.

We assume that the network is undirected and each node has it's own unique Id. There is an order defined on the set of possible Ids such that if x and y are both Ids, either $x > y$ or $y < x$. We assume that these unique Ids are sufficient in length such that random generation guarantees that no two Ids are equal.

4 Design Description

4.1 Overview

Our algorithm breaks down into two components. The first component is the construction of the spanning tree and the second component is the method for averaging the signals. These two processes run concurrently on each individual node.

4.2 Spanning Tree Construction

The self stabilizing spanning tree algorithm is based on a method by (Afek 1991). On a high level, every node in the network attempts to construct a tree in the network stemming from itself. Trees with roots that have larger node identities eventually overrun those with smaller node identities. Eventually, the tree with the largest node Id overruns all the trees. A node will leave it's tree if it detects a node in it's neighbor list whose root has a higher Id than the root of it's own tree. However, in order to join a new tree, the node must transmit request packets that get propagated up to the root of the tree after which the root will grant the request. To maintain self stabilization, we define certain conditions that will restart the algorithm if violated at any node.

We now describe the method in detail. Each node stores three groups of variables related to the spanning tree process: graph variables, tree variables, and communication variables. A variable associated with node

v. Is denoted by $v.variable$. For example, the Id variable of node v is denoted v.Id. The details of these variables is detailed below:

1. Graph Variables { Id, Edge_list}

   v.Id is the unique identification key of node v. It is assumed that these Ids are uniquely generated when a node attempts to join the network and that no node has an ID equivalent to another ID in the network.

   v.Edge_list is a list of node IDs in the network that correspond to the neighbors of node v.

2. Tree Variables { Root, Parent, Distance}

   v.Root is an ID that corresponds to the node ID that is the root of node v's current tree.

   v.Parent is the identity of a neighbor of v which is the parent of v in the tree.

   v.Distance is the distance from node v to the root of the tree. Each hop between a child and a parent node is a unit of 1.

3. Communication variables { Request, To, Direction, From}

   v.Root is an ID that corresponds to the node ID that is the root of node v's current tree.

   v.Parent is the identity of a neighbor of v which is the parent of v in the tree.

We now make the following definitions to aid in the presentation of the algorithm.

Definition 1. We say v is *a child of w* and that *w is a parent of v* if the following boolean condition is true: $(w.Id \in v.Edge\_list) \wedge (v.Parent = w.Id) \wedge (v.Root = w.Root) \wedge (v.Distance = w.Distance + 1)$

Definition 2. We say a node is a *root* if $(v.Root = v.Id) \wedge (v.Parent = v.Id) \wedge (v.Distance = 0)$

In a globally sound spanning tree, we would like it to be the case that at every node v, either v is the root of the tree or, there is a node in v's neighbor list that has the same root as v but is one distance unit less than v is to the root. Further we would like it so that no root in node v's edge list has a higher root Id than node v. We formalize this requirement with a boolean condition denoted C1:

Condition C1: $\{[(v.Root = v.Id) \wedge (v.Parent = v.Id) \wedge (v.Distance = 0)] \vee [(v.Root > v.Id) \wedge (v.Parent \in v.Edge\_list) \wedge (v.Root = (v.Parent).Root)]\} \wedge (v.Root \geq max_{x \in v.Edge\_list} x.Root)$

The aim of our algorithm is to detect violations of condition C1 and ensure that the network eventually returns to a legal global state. The formal description of the algorithm is given in table 1. The program consists of a eight boolean condtitions that are continuously checked at every node. If a given boolean expression is violated, the algorithm takes an appropriate step.

If a node notices a violation of C1, the node must make itself the root of a tree. In this case, C1 does not necessarily become true, by we define a new condition C1' that does become true.

Condition C1': $[(v.Root = v.Id) \wedge (v.Parent = v.Id) \wedge (v.Distance = 0)] \vee [(v.Root > v.Id) \wedge (v.Parent \in v.Edge\_list) \wedge (v.Root = (v.Parent).Root)]$

If both C1 and C1' are not both true at node v, this means there is a neighbor of v with a root with a higher Id than that of node v. Node v must then attempt to join that neighbor's tree. That node must request to join another tree (Action 2).

Formal presentation of self stabilizing spanning tree algorithm

At node v, do forever:

1. If: $C1 \wedge C1'$
   - Do: v.Root := v.Id; v.Parent := v.Id; v.Distance := 0;
2. If: $C1' \wedge (u.Root = max_{x \in Edge\_list} x.Root) > v.Root$
   - Do: v.Request := v.From v := v.Id; v.To := u.Id; v.Direction := Ask; v.Root := v.Id;v.Parent=v.Id; v.Distance := 0;
3. If: $C1 \wedge\ C2$
   - Do: v.Request := v.To := v.From=v.Direction := undefined
4. If: $C1 \wedge C2 \wedge\ C2' \wedge (\exists w \in v.Edge\_list \mid (w.Direction = Ask) \wedge (w.To = v.Id) \wedge (w.Request = w.Id) \wedge (w.Root = w.From))$
   - Do: v.Request := v.From :=w.id v.To := v.Parent; v.Direction := Ask
5. If: $C1 \wedge C2 \wedge C2' \wedge (\exists w \in v.Edge\_list \mid (w \text{ child of } v) \wedge (w.To = v.Id) \wedge (w.Request = w.Id = w.Root = w.From))$
   - Do: v.Request := w.Request; v.From := w.Id; v.To:= v.Parent; v.Direction:=Ask
6. If: $C1 \wedge C2' \wedge (visaroot) \wedge (v.Direction = Ask)$
   - Do: v.Direction =Grant
7. If: $C1 \wedge C2' \wedge (v.To = v.Parent = u.Id) \wedge (u.Direction = Grant) \wedge (u.Direction = Ask) \wedge (u.Request = v.Request) \wedge (u.From = v.Id)$
   - Do: v.Direction :=Grant
8. If: $C1' \wedge C1 \wedge (v.Direction = Ask) \wedge (v.Request = u.Request = v.From = v.Root = v.Id) \wedge (u.From = v.Id) \wedge (u.Direction = Grant) \wedge (v.To = u.Id)$
   - Do: v.Parent := u.id; v.Distance :=u.Distance+1; v.Root:=u.Root; reset communication variables

Explanation of algorithm steps

Step 1. If node v cannot find a neighbor with a higher root Id than it's own Id, it makes itself a root.

Step 2. If there is a neighbor of node v that has a higher root Id than node v's root Id, request to join that tree. Node v makes itself a root.

Step 3. If the global condition for a spanning tree holds at node v, but the communication registers have values in them, set them to undefined so that node v may service requests.

Step 4. If there is a node w in the neighbor list of node v that is requesting to join node v's tree, node v forwards node w's request to v.Parent

Step 5. If there is a child of node v that has a request, forward this request. Note, requests from children take priority over nodes from outside the tree.

Step 6. If node v is the root of a tree and it gets an Ask request, grant the Ask request.

Step 7. If node v's parent sends it a grant packet, forward packet to child.

Step 8. If node v recieves a grant packet and node v is the root of a tree, node v updates it's tree variables to agree with the new tree and reset's its communication variables.

4.3 Averaging Mechanism

We now present the method for averaging signals after the spanning tree is constructed. Although this method runs concurrently with the self stabilizing spanning tree protocol, it's correctness can only be guaranteed if the network is in a legal global state. We consider both constant and sinusoidal signals with known frequency but unknown amplitude and phase.

Constant Inputs

We present the following variable list:

- input: the input signal of node v. In the constant input case, this value is just a scalar in $\mathbb{R}$. This value does not change.
- cumtot: The cumulative sum of all input sums of node v's descendent's.
- Nchild: The total number of descendants of node v.
- reported: An indicator variable at node v which is 1 if all of node v's children have reported their values of cumtot and Nchild and 0 if there is a node that is a child of node v that has not reported it's average variables.
- Children: Children of node v. This variable will change at each time step. It must be updated at each turn.
- global_avg: Node v's current estimate of the global average of all signals

We present the psuedo-code for the algorithm below:

```
At node v:
Step 1. Make list of children
    for all nodes w in v.Edge_list
        if w is a child of v
            store w.Nchild, w.cumtot, and w.reported
        end if
    end for

Step 2. Check if all children have all information from descendants
   if (number of children of node v with w.reported=1)=(number of children of node v)
       continue
   otherwise
        exit
    end if

Step 3. Update global average
   if not root
        read global average from parent
   else
        compute global average by:
        global average=(cumtot+v.input)/(v.Nchild+1)
    end if
```

Sinusoidal signal For the sinusoidal case, we assume each node has carries a signal at the same frequency but the amplitude and phase of each node's input signal varies. Thus, each node carries a signal of hte form

$$v(t) = A\sin(\omega t + \phi) \tag{3}$$

. Note this can be rewritten as

$$v(t) = A_1 sin(\omega t) + A_2 cos(\omega t) \tag{4}$$

, where$\phi = arctan(\frac{A_2}{A_1})$ and $A = \sqrt{A_1^2 + A_2^2}$. We assume that each node is capable of obtaining estimates of phase and amplitude readings of their respective input signals through a mechanism such as moving least squares or kalman filtering. We also assume that these estimates are within an acceptable level of accuracy

of their true values. Each node then converts their estimates to the form in Eq. 4. It is now the case that we are simply tracking two inputs rather than one (A_1 and A_2). Assuming we can obtain the global average of these two values, we can obtain the global average of the signal by reconstructing Eq. 3. To do so, we run the same algorithm as for the constant input case for two constant inputs rather than one.

5 Performance Testing

We would like to understand how our algorithm performs in a variety of different situations. The rest of this paper is organized as follows: we elucidate the structure and construction of a variety of random networks. We then discuss simulation results in those random networks. Finally we examine the robustness of the spanning tree algorithm to communication failures.

5.1 Random Network Construction

Erdos Reyni

The erdos reyni model is the most basic model of a random network. The model sets a link between any two links in the graph with independent equal probability. If p is the probability of two nodes sharing a link, the graph G, on average has $\binom{n}{2}p$ links in its edge set. Varying the probability p, results in different levels of network connectivity. While erdos reyni networks are easy to construct, they have less structure of the other random networks we will consider. In our examination, we will investigate both the effect of number of nodes and link probability on the convergence of our algorithms.

Small World

Small world networks model networks where certain groups of nodes are more likely to link each other than other nodes in the network. We construct such a network by first constructing a ring of all the nodes. We then connect each node to its next nearest neighbors in the ring. We then take each link in the network, and reassign it with probability p. We will investigate small world network size as well as the effect of link reassignment probability on the system convergence.

Scale Free

Scale free networks simulate certain networks where there exist certain nodes that are "hubs." These networks tend to have exponential degree distribution. Their construction involves the mechanism of preferential attachment. We start with a network of three nodes. We then attach a node with a fixed initial degree. The links of the new node are chosen with a probability distribution relative to the current degree of each node in the network. Thus those nodes which start off with more connections eventually attract more nodes than those nodes which start off with fewer connections. We will investigate the changes in initial degree on settling time.

Nearest Neighbor

In many engineered networks, often times there are agents interacting in a given environment in which communication is regulated by distance. In this network, we assume all agents are in a k-cell in $\mathbb{R}^3$ whose coordinates are uniformly distributed. We then take the pairwise distance between all nodes in the network

and establish links between all agents within a distance of r of each other. We investigate the effect of this distance on settling time.

Summary of random network test cases

Erdos Reyni	Small World	Scale Free	Nearest Neighbor
network size	network size	network size	network size
link probability	reassignment probability	initial degree	cutoff distance

6 Results

Before examining any of the algorithm performance in the random network topologies, we would like to first develop an understanding as to the character of the response of these two algorithms to changing network topology. We first examine the convergence of these algorithms using a simple, 10 node erdos reyni network with fifty percent link density. Figure 1 shows the response of the two algorithms to nodes joining the network. The y axis gives the maximum error from the true average estimate among all nodes in the network. We can see that while the spanning tree algorithm does worse for a longer period of time, it converges to the true value quicker than does the PI estimator. This result, as we will see, does not hold in general. We can see from this plot that unlike the PI estimator, convergence of this method is not smooth, and tends to vary a lot before achieving the true average value.

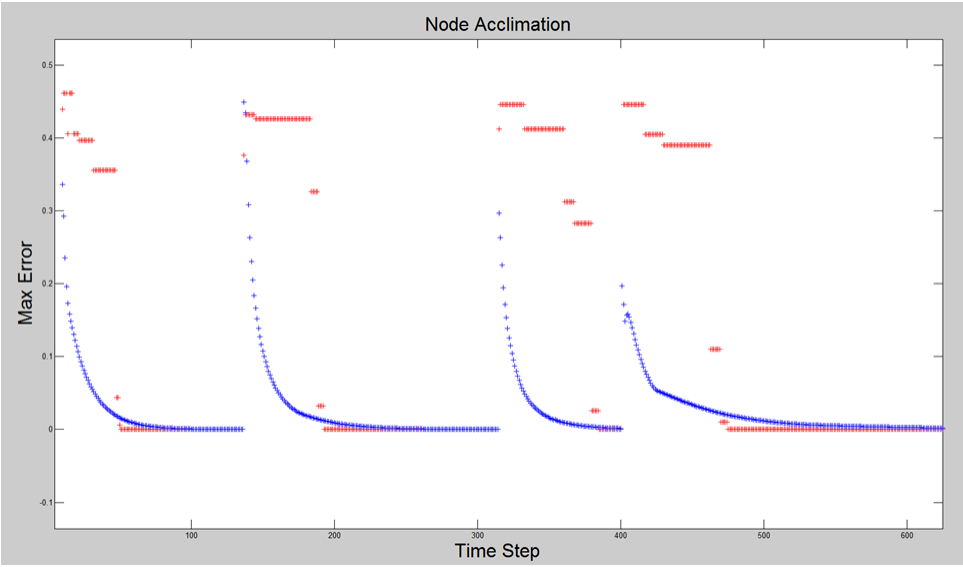

Figure 1: Adding nodes to an erdos reyni network, red=Spanning tree, blue= PI estimator

We now examine the robustness of our algorithm in the erdos Reyni network. Figure 2 shows the convergence time of our algorithm's as a function of network size. For all tests, we define convergence time as the time it takes for the maximum error in any node in the network to approach a value less than 10^-4. For this test we used a link probability that was constant at p=.5. The results show that for small network sizes, the spanning tree algorithm outperforms the PI estimator. For network sizes at around thirty nodes, the two algorithms converge in approximately the same amount of time. After this point however, we see the spanning tree method continue to scale with network size but the PI estimator level off at a value at around 100 time steps. From figure 1, we can conclude that in a Erdos Reyni network with 50% link density,

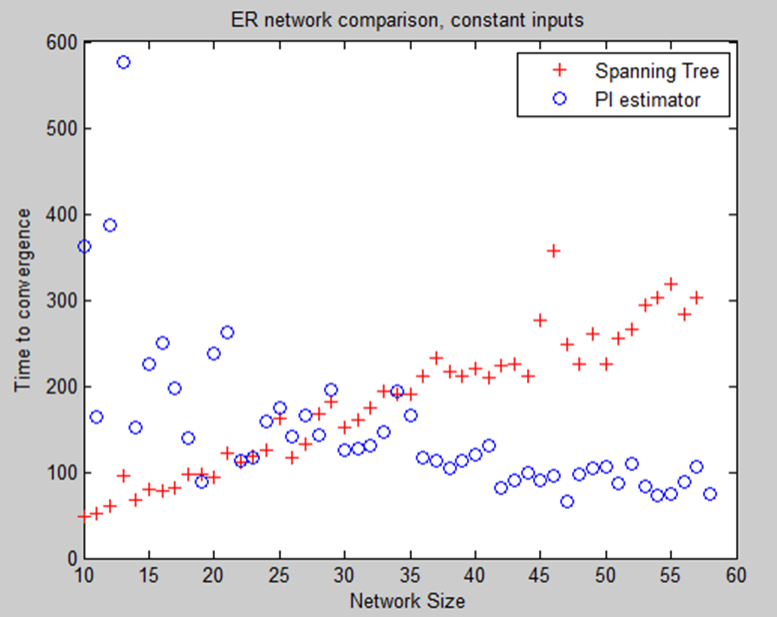

Figure 2: Erdos Reyni, Varying network size, link probability p=.5

the convergence of the spanning tree algorithm scales directly with network size. The PI estimator on the other hand, seems to do better with larger network size but reaches a threshold after which the performance improvement saturates. A plausible next question would be whether this trend holds for erdos Reyni networks of differing link densities. Figure 3 attempts to shed more light on this question. Each plot examines an erdos reyni network of a fixed size with varying link probability. We can see that for any given plot, the number of time steps that the spanning tree algorithm takes to achieve steady state remains fairly constant across networks with differing link densities. On the other hand, the PI estimator seems to take drasitcally less time to converge in networks with high link densities.

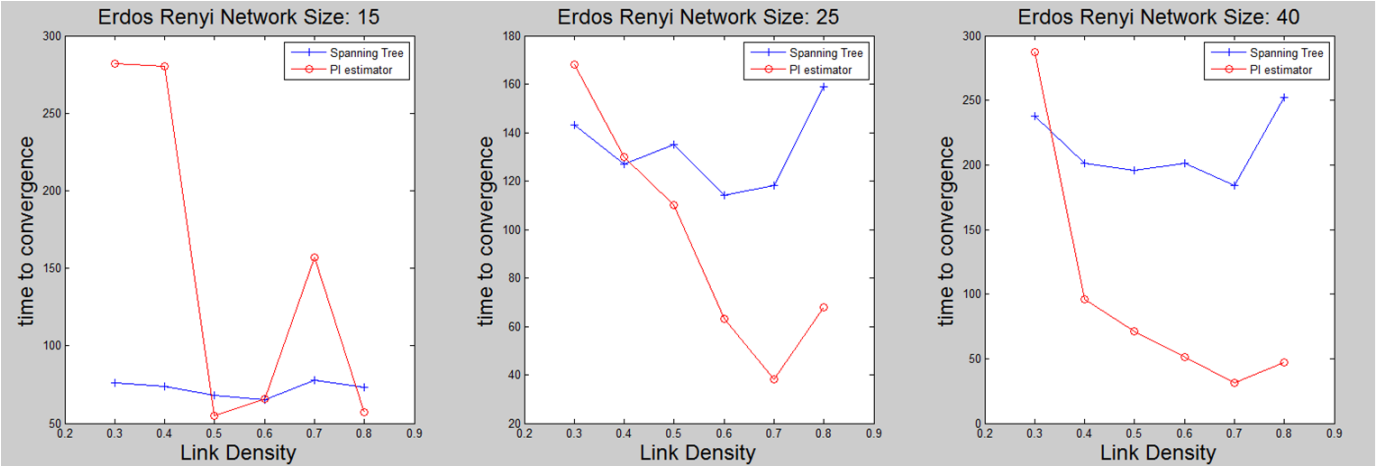

Figure 3: Erdos Reyni, Varying network size

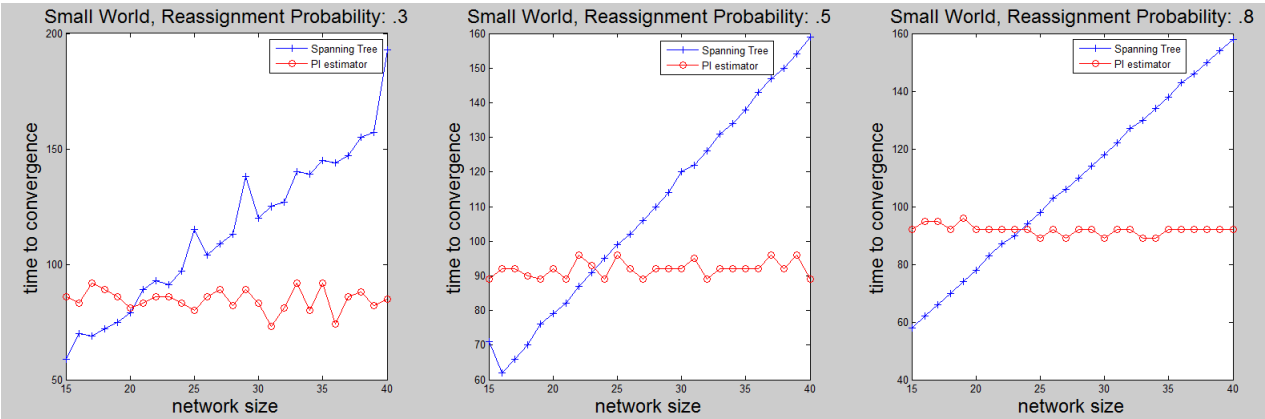

Figure 4: Small world network, Varying link reassignment probabilities

We now examine convergence performance of the algorithms in small world networks of differing size and link reassignment probability. Figure 4 shows three different link reassignment probabilities associated with a set of small world networks of increasing size. The algorithm shows that while the PI estimator drastically underperforms the spanning tree algorithm for low reassignment probability, for higher link reassignment probability, we see that the trend in both methods convergence time is unaffected by the link reassignment probability. The spanning tree method scales linearly in all three cases and the PI estimator remains fairly constant in all three cases. Nevertheless this result is illuminating. The link reassignment probability strictly affects the clustering of the networks. Low link reassignment corresponds to higher clustering. We see that while the spanning tree algorithm is linear for all levels of clustering,the higher clustering plot shows more variance in the linear relation than lower clustering plot. Furthermore, it is interesting to note that in all the networks we tested that were of a certain size, the number of links in the network remain constant. In light of the trend we see for the PI estimator this becomes very interesting. We saw previously that the PI

estimator was highly sensitive to network link density. In this case, the networks have relatively constant link density. This shows that the clustering in a network does not affect the PI estimator so much as the link density.

We now examine the results for the scale free network topology in figure 5. In the scale free network, we see that the PI estimator drastically under performs the spanning tree methodology. In this framework, the network has very low link density but we can see that by changing the initial degree of the node from one to three we achieve orders of magnitude of difference in performance (from 10^5 to 10^3). The spanning tree algorithm once again scales linearly with network size.

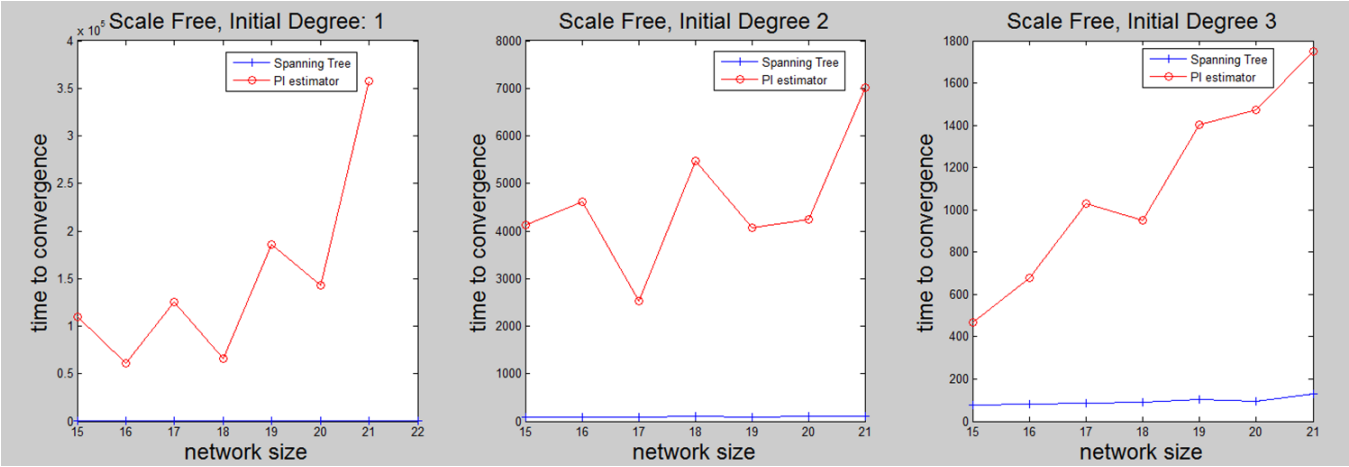

Figure 5: Scale free network, Varying levels of initial degree

We finally examine settling time in the nearest neighbor network. We varied the cutoff distance for link establishment to take on .1,.5,1,1.1,1.5 and 2.1. This serves once again to show that the PI estimator is insensitive to network size but sensitive to link density and the spanning tree algorithm behaves in the opposite manner. For cutoff distances less than .1 the network is very sparse. as we increase the cutoff distance, the network approaches the case of being a fully connected graph. In the fully connected graph (cutoff=2.1), we see that the spanning tree algorithm, behaves perfectly linearly and increases with network size and once again, the PI estimator is unaffected. This is because for every fully connected graph, the link density is the same.

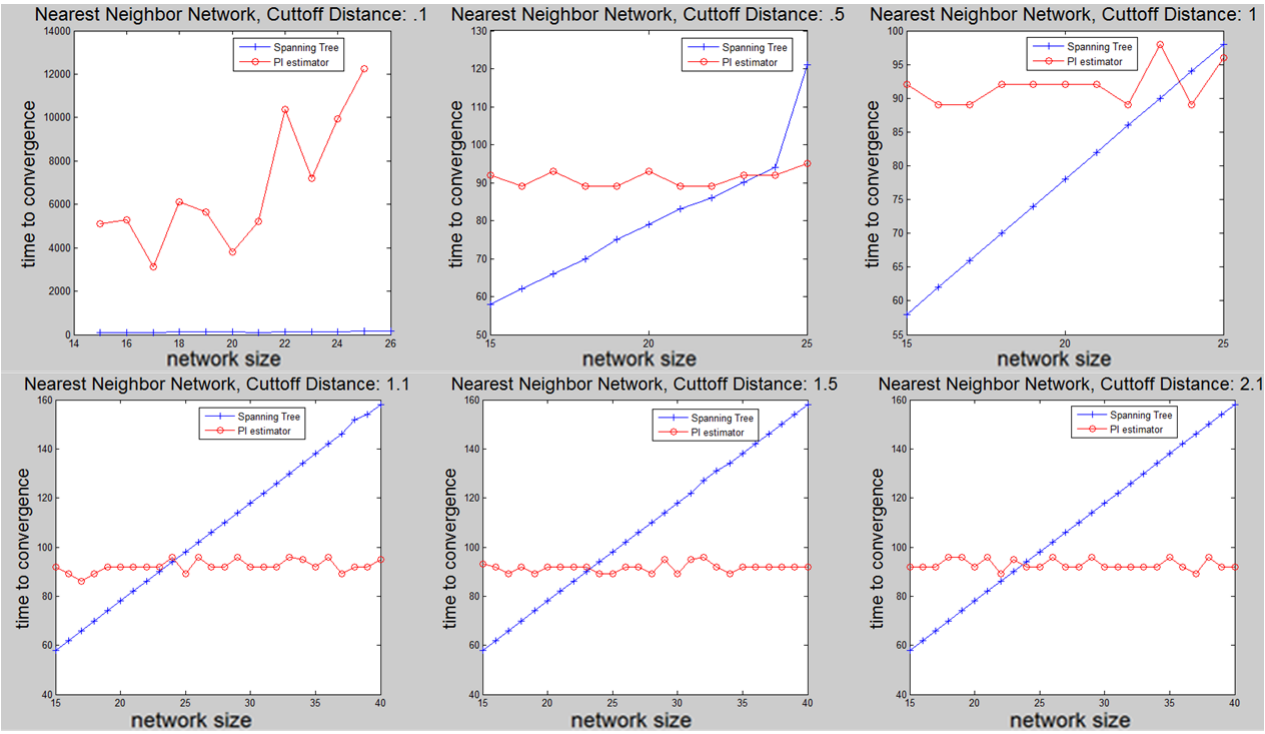

Figure 6: Nearest Neighbor Network, Varying levels of Cutoff Distance

We now test the robustness of our algorithm to communication failure. We construct varying erdos reyni networks with link density .5. We assume two types of communication failures of our algorithm. The first is a complete partial drop of information for one time step (i.e single packet loss). The second is a complete communication drop off. In order to simulate these two types of failure, we note that in the spanning tree algorithm, steps 4-8 require outside packet transmission. We simulate single packet loss by executing *each* of these steps with probability p. We simulate complete communication failure by executing *all* of these steps with probability p. For our testing, we take p=.8 to be the probability of failure.

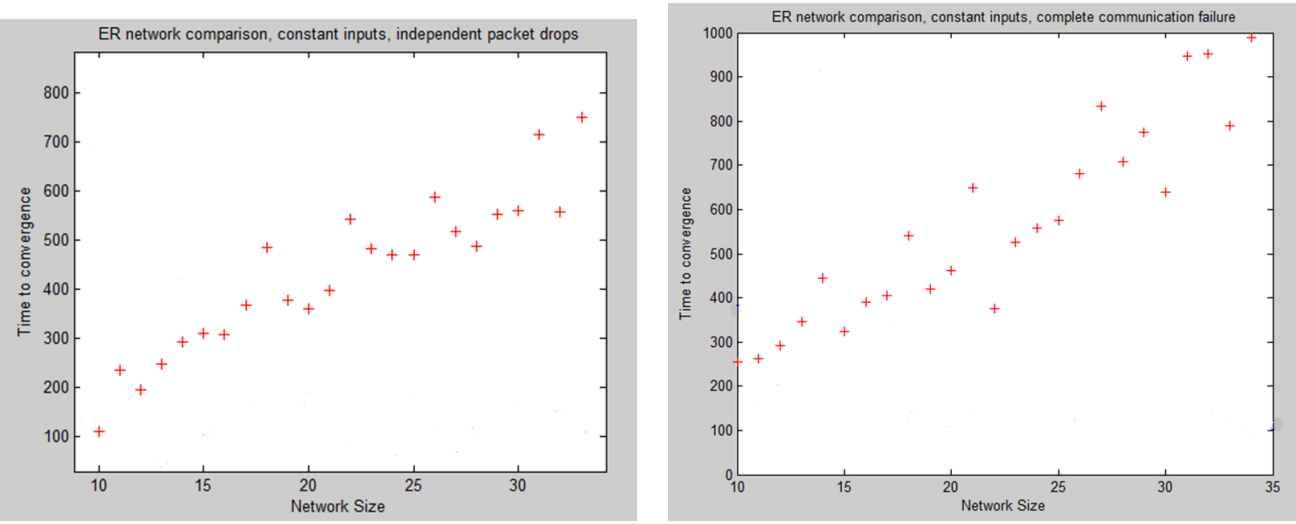

Figure 7: Communication failure, probability p=.8

We see from figure 7 that single node communication failure takes twice as long to converge as noiseless communication (see figure 2). Similarly, complete communication failure takes approximately three times as long to converge. Thus, an 80% probability of communication failure seems to only marginal reduction convergence time. Future work might attempt to simulate harsher communication conditions.

7 Broader considerations

We have observed the performance of our methodology in a variety of situations. We propose questions about practical application of this method in an engineered system. We have seen that spanning tree performance degrades with increasing network size and the PI estimator performance degrades with increasing. In practice we would like an algorithm that performs well in a wide variety of network topologies. One could imagine an engineered solution in which a hybrid approach is implemented. The simplest of which would be running both algorithms in parallel on an agent in a robotic network and choosing the answer that converges first. Of course, this would have memory and speed overhead associated with it. We also see that in the scale free network, convergence of the PI estimator performs very poorly in comparison to the spanning tree methodology. The structure of this network gives way to an alternative control mechanism that would probably make the PI estimator a more feasible design option. Scale free networks have certain nodes that act as "hubs"– meaning they have high degree relative to other nodes in the network. One could imagine these nodes acting as centralized controllers for the neighborhoods they connect to.and the implementing a spanning tree or PI approach on the network of hubs.

8 Conclusions and Future Work

In this paper we have proposed a newly designed method of robust average consensus in a distributed system. The algorithm works for both constant and sinusoidal signals with known frequency but unknown amplitude and phase. We have empirically demonstrated the robustness of this algorithm, examined it's performance in a variety of complex networks, and elucidated conditions in which it out performs the standard PI estimator. We show that where each of the two algorithms fails the other one succeeds. One can say that in general, the PI estimator works better in networks with high link densities. The spanning tree algorithm works better in networks with fewer nodes. The PI estimator settling time is intimately tied to the eigenvalues of the graph laplacian. The second smallest eigenvalue of the graph laplacian determines the rate of settling time of the PI estimator. It is theorized that there are characteristically different expectations for the eigenvalue set in each model of random network generation. A possible direction of future study would be to determine characteristic differences in the eigenvalue set of different graph laplacians affect PI settling time. We can analytically show that the eigenvalues of the graph laplacian of a completely connected graph are equal. It would be interesting to see how deviations from this case affect the eigenvalues of the Laplacian. There were many assumptions in our model that may or may not hold true in a real system. One example of particular note is our assumption that all clocks are synchronized to a centralized clock. If we relax this assumption,

our algorithm for the sinusoidal case would have to change. In particular, we would need a way of either computing the global phase in a way that is independent of a local time stamp or a way to synchronize clocks on the network. The convergence rates tested so far was all for constant input signals. An obvious next step would be to extend the analysis done here to sinusoidal signals.

9 References

Afek, Y. K., Shay, Y. M. , (1991). Memory-efficient self stabilizing protocols for general networks. Proceedings of the 4th international workshop on distributed algorithms, London, UK.

Bai, H., Freeman, R. A., Lynch, K. M. (2010). 2010 49th IEEE conference on decision and control. IEEE.

10 List of Figures

List of Figures