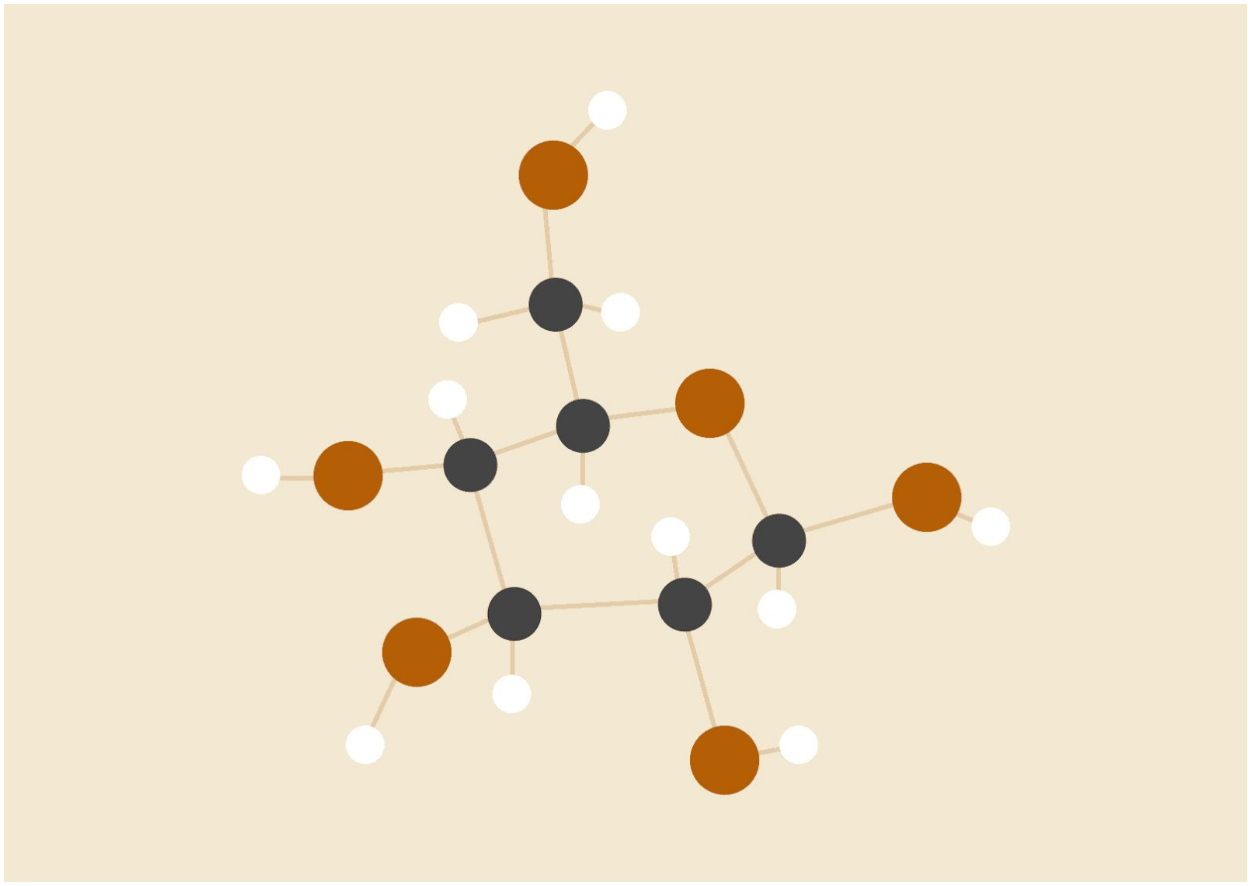
CSP 290 REPORT

Consensus Propagation



Mukesh Singh Pokharia
Navdeep Singh
Vineet Bakshi

Dec 2020

1st Sem, 2020

ABSTRACT

In this project, we analysed the algorithm proposed in (Moallemi & Roy, 2006). It proposes consensus propagation, an asynchronous distributed protocol for averaging numbers across a network, which we have simulated synchronously. Consensus propagation can be viewed as a special case of belief propagation(Dai & Zhang, 2007).

In particular, beyond singly-connected graphs, there are very few classes of relevant problems for which belief propagation is known to converge. But consensus propagation converges also for general graphs, while the time of convergence can only be commented on d-regular graphs (after diameter no. of rounds).

The tree algorithm converges perfectly for spanning tree of social graphs, while the graph algorithm converges along with some variance from the mean. We observed that variance in convergence increases with the increase in connectivity of the graph.

INTRODUCTION

Problem Statement: Consider a network of n nodes in which the i_{th} node observes a real number $y_i \in R$ and aims to compute the average of all Yi's. The purpose is to reach the best possible estimate for all nodes, through local processing and information exchange over the network. This is a type of **consensus estimation** problem.

Relevance of the problem: Recent advances in information technology are leading to a paradigm shift towards ad hoc networking, distributed processing and pervasive computing and communications. Examples include mobile ad hoc networks, wireless mesh networks, and sensor networks for various military, commercial, environmental and emergency applications. In both wireless sensor and peer-to-peer networks, for example, there is a need for simple protocols for computing aggregate statistics, and averaging enables computation of several important ones. For example, (Xiao et al., 2005) considers an averaging problem with applications to load balancing and clock synchronization. Policy-gradient-based methods for distributed optimization of network performance rely heavily on averaging protocols.

(Moallemi & Roy, 2006, Pg 1) proposes a new **asynchronous** protocol, **consensus propagation** as a solution to the above problem. This is a **distributed** protocol because computations at each node require only information that is locally available to the node, no further information about the network topology is required. No central entity which can contact each node exists. It is an **asynchronous** protocol because only a subset of the potential messages is transmitted at each time. (Moallemi & Roy, 2006, 5) provides a proof of convergence for this protocol even when executed asynchronously and provides a bound for the convergence time in case of regular graphs. Consensus propagation can be viewed as a special case of belief propagation, and it contributes to the belief propagation literature(Dai & Zhang, 2007).

Some previously proposed solutions are **probabilistic counting** and **pairwise averaging**. (Moallemi & Roy, 2006) proves convergence time of consensus propagation to scale more gracefully with the number of nodes than pairwise averaging, and for certain classes of graphs, quantifies the improvement. In terms of convergence rate, probabilistic counting dominates consensus propagation in the asymptotic regime. However, consensus propagation is likely to be more effective in moderately-sized networks (up to hundreds of thousands or perhaps even millions of nodes).

PROBLEM DESCRIPTION

Consider a connected undirected graph (V, E) with $V = \{1, . . . , n\}$. For each node $i \in V$, let $N(i) = \{j \mid (i, j) \in E\}$ be the set of neighbors of i. Let $\boldsymbol{E} \subseteq V \times V$ be a set consisting of two directed edges {i, j} and {j, i} per undirected edge (i, j) ∈ E. (In general, we will use braces for directed edges and parentheses for undirected edges.)

Each node $i \in V$ is assigned a number $y_i \in R$(for simplification purposes we have considered y_i equal to i). The goal is for each node to obtain an estimate of the average of all y_i`s, through an asynchronous distributed protocol in which each node carries out simple computations and communicates parsimonious messages to its neighbours.

SOLUTION DESCRIPTION (CONSENSUS PROPAGATION)

(Moallemi & Roy, 2006) proposes consensus propagation as an approach to the aforementioned problem. In this protocol, if a node i communicates to a neighbour j at time *t*, it transmits a message consisting of two numerical values. Let $\boldsymbol{\mu}_{ij} \in R$ and $\mathbf{K}_{ij} \in R^+$ denote the values associated with the most recently transmitted message from *i* to *j* at or before time *t*. At each time *t*, node *i* has stored in memory the most recent message from each neighbor: $\{\boldsymbol{\mu}_{ui}, \mathbf{K}_{ui} \mid u \in N(i)\}$. If, at time $t+1$, node i chooses to communicate with a neighboring node $j \in N(i)$, it constructs a new message that is a function of the set of most recent messages $\{\boldsymbol{\mu}_{ui}(t), \mathbf{K}_{ui}(t) \mid u \in N(i)|j\}$ received from neighbors other than *j*.

In the sections below we describe the case of a tree and a general graph as dealt in (Moallemi & Roy, 2006).

CASE 1 - TREEs

Tree is an acyclic connected graph. In the case of trees, the consensus propagation algorithm converges perfectly(with no standard deviation) to the actual mean in diameter number of iterations.

THEORY

Tree is the special case of a singly-connected graph. That is, a connected graph where there are no loops present. Assume, for the moment, that at every point in time, every pair of connected nodes communicates. As illustrated in **Fig. 1**, for any edge {i, j} ∈ **E**, there is a set $S_{ij} \subset V$ of nodes, with $i \in S_{ij}$, that can transmit information to $S_{ji} = V \setminus S_{ij}$, with $j \in S_{ji}$, only through {i, j}.

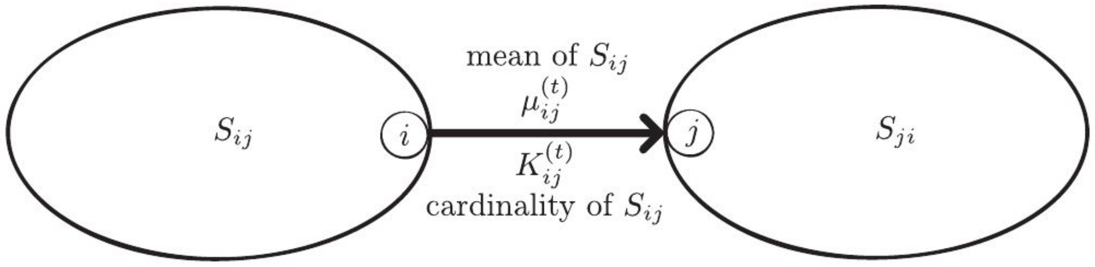


Fig. 1. Interpretation of messages in a singly connected graph with $\beta = \infty$.

In order for nodes in S_{ji} to compute y, they must at least be provided with the average μ^*_{ij} among observations at nodes in S_{ij} and the cardinality $K_{ij}^* = |S_{ij}|$. Similarly, in order for nodes in S_{ij} to compute y, they must at least be provided with the average μ^*_{ji} among observations at nodes in S_{ji} and the cardinality $K_{ji}^* = |S_{ji}|$. These values must be communicated through the link {j, i}. The messages μ_{ij} and K_{ij} , transmitted from node i to node j, can be viewed as iterative estimates of the quantities $\mu^*_{ij}(t)$ and $K_{ij}{}^*(t)$.

$$\mu_{ij}^{(t)} = \frac{y_i + \sum_{u \in N(i)\setminus j} K_{ui}^{(t-1)} \mu_{ui}^{(t-1)}}{1 + \sum_{u \in N(i)\setminus j} K_{ui}^{(t-1)}}, \quad \forall \{i,j\} \in \vec{E}, \quad \text{(1a)}$$

$$K_{ij}^{(t)} = 1 + \sum_{u \in N(i)\setminus j} K_{ui}^{(t-1)}, \qquad \forall \{i,j\} \in \vec{E}. \quad \text{(1b)}$$

For corresponding code refer to line 19-37 (CodeFile--consensus.py)

At each time t, each node i computes an estimate of the global average y according to

$$x_i^{(t)} = \frac{y_i + \sum_{u \in N(i)} K_{ui}^{(t)} \mu_{ui}^{(t)}}{1 + \sum_{u \in N(i)} K_{ui}^{(t)}}.$$

For corresponding code refer to line 54-59, 79 (CodeFile--consensus.py)

Assume that the algorithm is initialized with K(0) = 0. A simple inductive argument shows that at each time t ≥ 1, μ_{ij}(t) is the average among observations at the nodes in the set S_{ij} that are at a distance less than or equal to t from node i. Furthermore, K_{ij}(t) is the cardinality of this collection of nodes. Since any node in S_{ij} is at a distance from node i that it at most the diameter of the graph, if t is greater than the diameter of the graph, we have K (t) = K^* and μ(t) = μ^* .Thus, for any $i \in V$, and t sufficiently large,

$$x_i^{(t)} = \frac{y_i + \sum_{u \in N(i)} K_{ui}^{*} \mu_{ui}^{*}}{1 + \sum_{u \in N(i)} K_{ui}^{*}} = \overline{y}.$$

OBSERVATION

So, x_i(t) , for each node i ,converges without any deviation to the global average y_{mean}. Further, this algorithm converges in as short a time as is possible, since the diameter of the graph is the minimum amount of time for the two most distant nodes to communicate.

CASE 2 - GENERAL GRAPH

Any set of vertices and edges denoted by *G(V,E)*.

THEORY AND THE ALGORITHM

Consider a graph with cycles. For any directed edge {i, j} ∈ **E** that is part of a cycle, $K_{ij}(t) \to \infty$. Hence, the algorithm does not converge. A heuristic fix might be to compose the iteration *(1b)* with one that attenuates:

$$\tilde{K}_{ij}^{(t)} \leftarrow 1 + \sum_{u \in N(i) \setminus j} K_{ui}^{(t-1)},$$

$$K_{ij}^{(t)} \leftarrow \frac{\tilde{K}_{ij}^{(t)}}{1 + \tilde{K}_{ij}^{(t)}/(\beta Q_{ij})}.$$

For corresponding code refer to line 42-43 (CodeFile--consensus.py)

Here, $Q_{ij} > 0$ and $\beta > 0$ are positive constants. We can view the unattenuated algorithm as setting $\beta = \infty$. In the attenuated algorithm, the message is essentially unaffected when $\tilde{K}_{ij}(t)/(\beta Q_{ij})$ is small but becomes increasingly attenuated as $\tilde{K}_{ij}(t)$ grows. This is exactly the kind of attenuation carried out by consensus propagation. Understanding why this kind of attenuation leads to desirable results is a subject of (Moallemi & Roy, 2006).

Consensus propagation is parameterized by a scalar $\beta > 0$ and a non-negative matrix $Q \in R_+^{n \times n}$ with $Q_{ij} > 0$ if and only if $i \neq j$ and $(i, j) \in E$. For each $\{i, j\} \in E$, it is useful to define the following three functions:

$$\mathcal{F}_{ij}(K) = \frac{1 + \sum_{u \in N(i) \setminus j} K_{ui}}{1 + \frac{1}{\beta Q_{ij}} \left(1 + \sum_{u \in N(i) \setminus j} K_{ui}\right)}, \tag{2a}$$

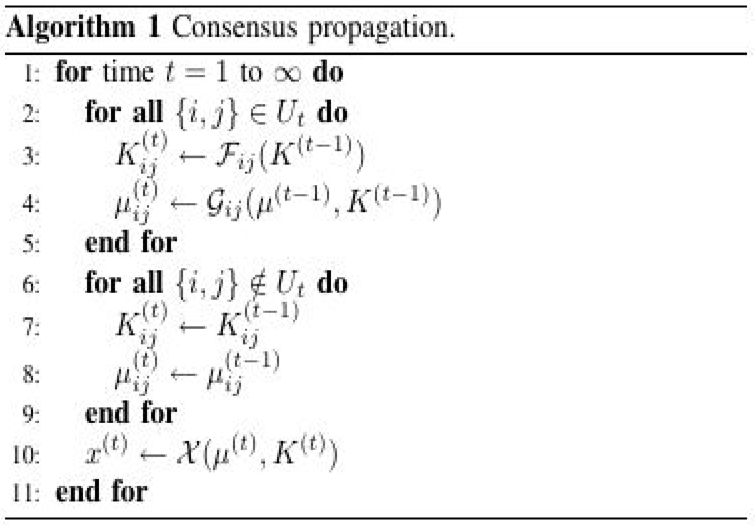
$$\mathcal{G}_{ij}(\mu, K) = \frac{y_i + \sum_{u \in N(i) \setminus j} K_{ui} \mu_{ui}}{1 + \sum_{u \in N(i) \setminus j} K_{ui}}, \tag{2b}$$

$$\mathcal{X}_i(\mu, K) = \frac{y_i + \sum_{u \in N(i)} K_{ui} \mu_{ui}}{1 + \sum_{u \in N(i)} K_{ui}}. \tag{2c}$$

For corresponding code refer to line 27,send_message function (CodeFile--consensus.py)

- $\varkappa_i$ represents the newly calculated value to be stored in the node itself.
- G_{ij} is the message which i will send to j.
- F_{ij} is the calculated cardinality by taking in β and Q factors.
- For simplification, we are considering $Q_{ij} = 1$.

For graphs, consensus propagation is presented below as Algorithm 1.



Algorithm 1 Consensus propagation.

```
1:  for time $t = 1$ to $\infty$ do
2:      for all $\{i, j\} \in U_t$ do
3:          $K_{ij}^{(t)} \leftarrow \mathcal{F}_{ij}(K^{(t-1)})$
4:          $\mu_{ij}^{(t)} \leftarrow \mathcal{G}_{ij}(\mu^{(t-1)}, K^{(t-1)})$
5:      end for
6:      for all $\{i, j\} \notin U_t$ do
7:          $K_{ij}^{(t)} \leftarrow K_{ij}^{(t-1)}$
8:          $\mu_{ij}^{(t)} \leftarrow \mu_{ij}^{(t-1)}$
9:      end for
10:     $x^{(t)} \leftarrow \mathcal{X}(\mu^{(t)}, K^{(t)})$
11: end for
```

For each t, denote by $U_t \subseteq \mathbf{E}$ the set of directed edges along which messages are transmitted at time t. When implementing synchronously, $U_t = E$ (because message will be transmitted across every edge when transmitting synchronously), so $\{i,j\} \notin U_t$ is null. So,Pseudo code from lines 6 to 9 are basically asynchronous implementations of the algorithm.
While in our synchronous simulation of this algorithm,only lines 1-5 and line 10-11 are of significance.

Consensus propagation is a distributed protocol because computations at each node require only information that is locally available.
In particular, the messages $K_{ij}(t) = F_{ij}(K(t-1))$ and $\mu ij(t) = Gij(\mu(t-1), K(t-1))$ transmitted from node i to node j depend only on $\{\mu_{ui}(t-1), K_{ui}(t-1) \mid u \in N(i)\}$, which node i has stored in memory. Similarly, $x_i(t)$, which serves as an estimate of y, depends only on $\{\mu_{ui}(t), K_{ui}(t) \mid u \in N(i)\}$.
In general, Consensus propagation is an asynchronous protocol which deals with only a subset of the potential messages being transmitted at each time.

In our implementation, we focused on the synchronous version of consensus propagation. This is where

$U_t = \vec{E}$ for all t. Note that synchronous consensus propagation is defined by:

$$K^{(t)} = \mathcal{F}(K^{(t-1)}), \quad (3a)$$

$$\mu^{(t)} = \mathcal{G}(\mu^{(t-1)}, K^{(t-1)}), \quad (3b)$$

$$x^{(t)} = \mathcal{X}(\mu^{(t-1)}, K^{(t-1)}). \quad (3c)$$

Challenges Faced And Experimentations ---

1. It is strenuous to hard-code every graph you want to run your simulation on, especially when your graph has a large number of nodes. So, the idea was to experiment on publicly available social network graphs, but all social networking graphs are available in .gml format.
   Therefore, we had to implement our own parser for the gml file which while parsing the file also converts the graph given in the gml file to our python graph data structure. For that we have used recursive descent parsing techniques to parse the file.

2. We also implemented our own interval graph generator because it was very hard to find an interval graph's dataset on the internet. So we made our own script which takes the number of nodes and clique-no of the graph as arguments and gives you the gml file of the respective interval graph. Then that gml file is parsed by our own recursive descent parser to feed it to our implementation of the consensus propagation.

3. Now sometimes the social networking graphs are not connected , meaning they have various islands. So, to test on those dataset we implemented functions which could connect those islands with a minimum number of edges required to do so.

4. Also we implemented a function to convert graphs (connected or unconnected) into a spanning tree to test our tree algorithm simulation of consensus propagation.

For generating spanning trees and connecting islands (of the graph) we used kruskal's algorithms because it was a reasonable choice for this purpose, given the fact that all gml files contain data about each edge only and kruskal's algorithm also works on the set of edges of graphs. We didn't have to make any adjacency list from the gml file to get it spanning tree, we just directly fed the data from the parser to kruskal's algorithm which gives us the spanning tree and if required made the graph connected.

5. To make an unconnected graph connected, we used kruskal's algorithm exploiting the fact that at the termination of kruskal algorithm it gives you the set of vertices, which are connected vertices and if the graph is not connected it will give you a list of sets containing different islands of the graph.
   Now to make the graph connected, we just need to connect these sets which could be easily done by taking one vertex A from one set and then connecting it to one vertex xi from each of the remaining sets.

6. For Kruskal's algorithm we need a disjoint set data structure that we implemented from scratch to reduce the overhead because we only wanted a few functionality of the specific data structure.

Observations

1. The proposed algorithm converges for general graphs but the iterations taken to converge can’t be determined for every graph.(can only be upper bounded for d-regular graphs)
2. The number of iterations is equal to the diameter in the case of tree graphs.
3. Standard deviation for the predicted mean increases as the connectivity in the cycle increases.(shown via experiment done by increasing clique-no in interval graph).
4. The deviation depends less on the number of nodes and more on the dense cyclic connectivity of the graph.(when clique-no increases from 3 to 4, the connectivity also becomes dense.)

   As is clearly visible by the graphs below------

   - When clique-no=3 , standard deviation increases very little(0 to 0.08) on increasing the number of nodes from 50 to 1000.
   - While when clique-no = 4, std.Deviation has a much larger magnitude(5units) even at a graph consisting of 50 nodes.

- Magnitude of std.Deviation varies from 5 to 20units for when the number of nodes varies from 50 to 1000.

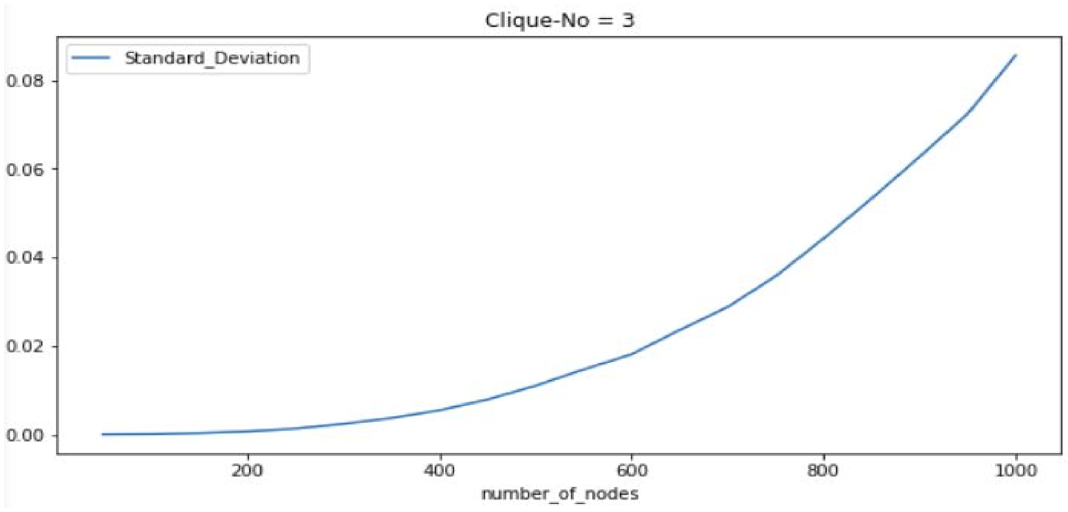


Figure 2

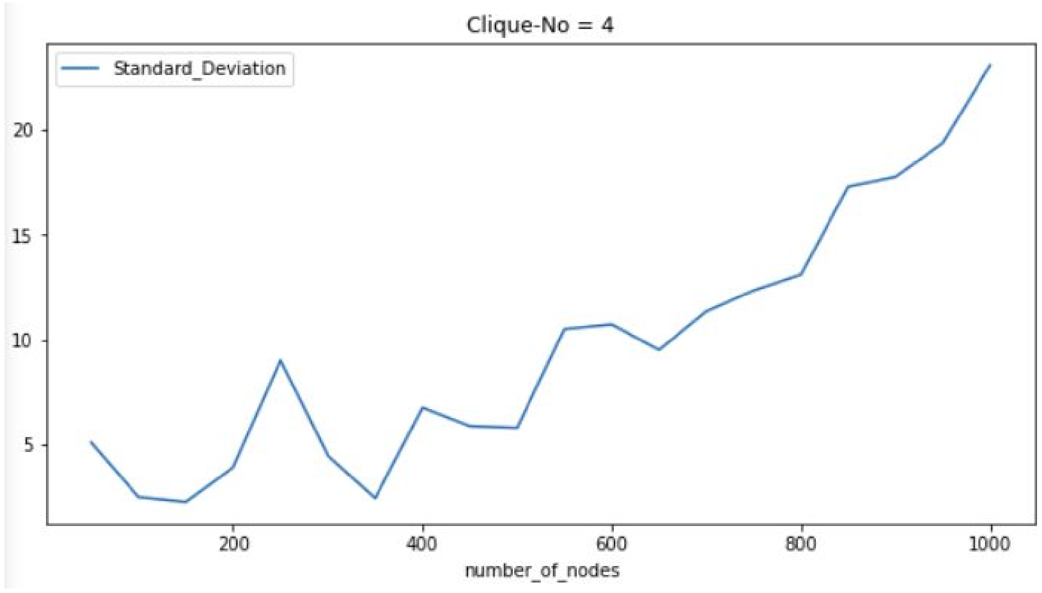
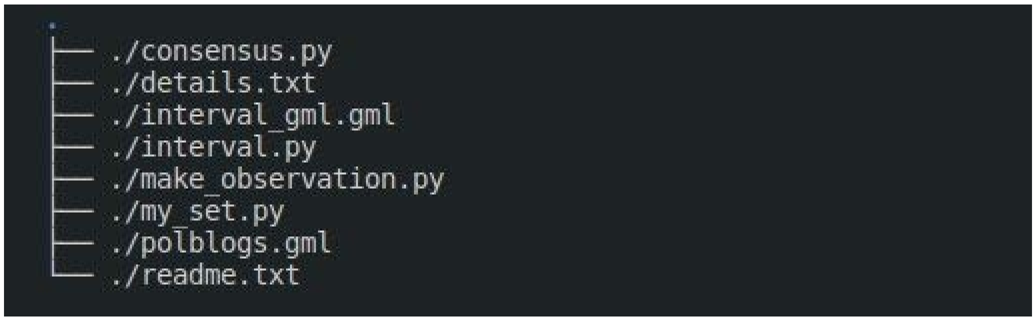


Figure 3

The above plots are experimental results in two different cases of interval graphs.(when clique-no is three and when clique-no is four).
For simplification purposes, if total no. of nodes = k, then values in the nodes vary from 1 to k, therefore the actual mean is ------(k)*(k+1)/2.

APPENDIX

This is the file structure we used to implement the above-mentioned algorithm.



```
.
├── ./consensus.py
├── ./details.txt
├── ./interval_gml.gml
├── ./interval.py
├── ./make_observation.py
├── ./my_set.py
├── ./polblogs.gml
└── ./readme.txt
```

Figure 4

(i) consensus.py: This file contains the main algorithms. In this file we implemented the simulations of two algorithms for consensus propagation one is for the tree graph and the other one is for the general graph. To implement these algorithms first we created a node class which has some data members required for the algorithm like each node knows its neighbouring node for message exchange, a mean variable to calculate the overall mean, a variable called cardinality which contains data regarding the neighbours. After implementing both algorithms, we make some helper functions which can be used to view the data in a nice format. This file takes input from a graph written in a .gml file and another argument tree/graph to ask which algorithm you want to simulate.

(ii) my_set.py : This file implements a compiler to parse a general .gml file to emit a graph data structure (called nodes). This file also has a function to calculate a spanning tree from the .gml file. This file also adds edges to make the graph connected, if required. It will connect different islands of the graph to give you a connected graph. To extract spanning tree and connected graphs we used Kruskal's algorithm.
Involved member functions:

1) give_graph
2) give_connected_graph
3) give_spanning_tree

(iii) make_observation.py : This file runs the algorithms for interval graphs of different clique-no and a particular number of nodes. All the observations corresponding to which are printed out in the details.txt file.

(iv) interval.py: This file contains the code for generating a .gml file corresponding to an algorithmically generated interval graph. It takes in two configurable parameters--- the clique-no of the graph and the number of nodes for your gml graph. The generating algorithm restricts the number of mutually connected nodes to the clique-no(parameter) specified.

(v) interval_gml.gml and polblogs.gml : These files contain the graphs in gml form that we run our algorithms on.

Instructions for running our Implementation!

Points to note---

- Each node has a unique identification namely--- node_id.
- Each node contains some value, the mean of which needs to be calculated.
- For convenience, we have taken value = node_id, for each such node.
- So, if no.of_nodes = 1000 nodes(that is node_id varies from 1 to 1000)
- Then, the value contained in the node will also vary from 1,2,3,4....,998,999,1000.
- Hence, in this particular example

  mean = summation(1 to 1000)/1000

  = 500.5
- **csp290.py contains the main algorithm. 2 configurable things in this file.**

  Usage:
  ```
  python3 consensus.py <gml_file> <graph/tree>
  ```
  Example:
  ```
  python3 consensus.py interval_gml graph
  ```
  Explanation:

  <graph> means using graph algorithm

  <tree> means using tree algorithm

- **interval.py is used to generate a gml file for interval graph (generated_file_name="interval_gml.gml")**

  Usage:
  ```
  python3 interval.py <clique-no> <no_of_nodes>
  ```
  Example:
  ```
  python3 interval.py 4 1000
  ```

Suggestion:

try to keep clique-no not larger than 10.

Explanation:

interval_gml.gml file is completely re-written, every time this command is executed.

- **my_set.py contains below 3 functions which parse the gml file to return their respective outputs.**
  1) give_graph
  2) give_connected_graph
  3) give_spanning_tree

- **make_observation.py is used only for interval_graph cases, to report the trend of clique-no vs no.of_nodes.**
  - It returns the output in a details.txt file.
  - Each time you run make_observation.py, output is appended in details.txt file.
  - To create a completely separate details.txt file, delete the old one first.

- **Running order**-------For making observation only
  - run make_observaton.py

- **Running order**----For checking algorithm (each result)
  - If you already have the gml file run csp290.py directly.
  - Else first run interval.py to create a interval_gml.gml file(interval graph) and then csp290.py

Github Link to code -> https://github.com/vineetb95/CSP290-Project

Bibliography

Dai, H., & Zhang, Y. (2007). Consensus Estimation via Belief Propagation. *2007 41st Annual Conference on Information Sciences and Systems, Baltimore*.

Moallemi, C. C., & Roy, B. V. (2006, Mar 19). Consensus Propagation. *IEEE Transactions on Information Theory, 2006,, VOL. 52*(NO. 11), 19.

Xiao, L., Boyd, S., & Kim, S. J. (2005, May). *Distributed average consensus with least-mean-square deviation*. preprint