Fire Detection System: Distributed Network

EE4540: Distributed Signal Processing

Erik Hagenaars
4272404

I. Introduction

In this report, a fire detection system will be implemented. The system will exist of multiple sensors with a specific communication radius. During this report Distributed Signal Processing problems and solutions will be studied. First, the design of the network will be designed. Secondly decentralized averaging algorithms will be evaluated. And lastly, a fire detection system will be implemented and tested using models.

II. Problem Description

Fires are always a problem in de industrialized sector. The response time and robustness of such a system is crucial to minimize the damage cost. In order to quickly detect a fire, multiple sensors will be needed. Since most industrial plants are large or have a substantial amount of interfering machines, only a short communication path will be possible. This problem should be solved using the theory of Distributed Signal Processing [1].

For this report, we model a plant with dimensions 30×60 meters. The sensors are able to communicate with other sensors with a maximum distance of 3 meters. The main goal is to detect a fire and communicate where the fire originated. The normal temperature will be a model with a fixed standard deviation σ and a mean μ which follow the Gaussian distribution $X \sim N(\mu, \sigma^2)$. The same distribution applies for the fire temperature. The only difference between the two distributions is the mean temperature. Normal temperature has a mean temperature of $20°C$ while the fire mean temperature is $80°C$. The distribution share the same deviation.

Specifically, the objective can be divided into three sections. In Section III, multiple network designs will be evaluated and picked. In Section IV, different averaging algorithms will be studied and tested. In Section V, the fire detection system will be proposed and performance will be evaluated. At the end, a small overall evaluation and conclusions will be given among with a discussion about possible future improvements or research.

III. Network Design

The first part of the research is creating and implementing the network. There are many types of different network topologies, but in this report two simple topologies will be evaluated. Firstly the random geometric network, which will be discussed in Section III-A. And secondly, the mesh network topology will be discussed in Section III-B. Specifically the square mesh network will be evaluated.

A. Random geometric network

The first graph proposal is the random geometric graph. The graph consists of N nodes. To create this graph, N uniform random points with the range $x, y \in [0, 1]$ are chosen as the location of the nodes. These nodes are connected when the distance to another node is less than r which is predefined by the algorithm.

Because this algorithm does not return a connected graph by default, r needs to be chosen carefully. One of the properties of the random geometric graph is that the probability that the graph is connected when $r^2 \geq \frac{2 \log n}{n}$ is given by $P(G = connected) = 1 - \frac{1}{n^2}$ [2]. This property holds for the range of zero to one.

To test this property on the dimensions of the problem, the number of nodes versus the number of connected components are plotted in Fig. 1. It is clear that for bigger number of nodes, the number of connected components declines. During our evaluations, the radius is fixed at 3 meter as stated in the specifications. It was found that at an average of 400 sensors were needed to create a connected graph.

B. Square mesh network

The mesh networks are very common among distributed networks. Most of the mesh networks are symmetric and almost all nodes have the same property. In this report, the square mesh network is evaluated because of simplicity. With a radius of 3 meters and a starting point at $(3, 3)$ we can simply calculate the number of nodes needed for the network. For the dimensions of the problem approximately $19 \times 9 = 171$ nodes are needed to uniformly distribute the plant while having a connected graph. The generation of this network is very simple.

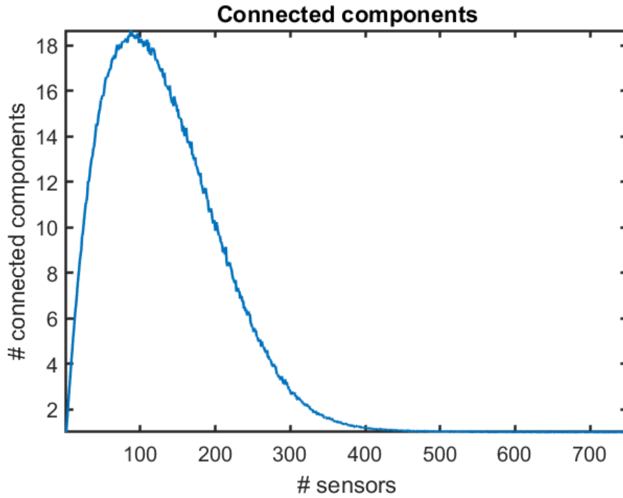

Fig. 1: Number of connected components of a random geometric graph for different numbers of nodes with a radius of 3m.

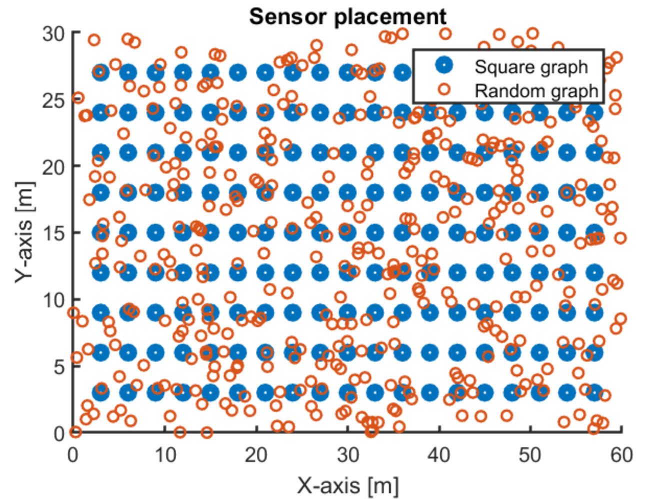

Fig. 2: Graph generation

TABLE I: Graph statistics.

Graph	Random	Square d = 3	Square d = 2.5	Square d = 2
Nodes	400	171	253	406
Average Degree	5.79-5.99	3.67	3.73	7.37
Longest Path	35-46	78	32	28

C. Results

The results of the graph generation can be seen in Fig. 2. The results show the graph generation of the random geometric graph and the square graph where nodes are placed 3 meters apart. Table I shows the statistics of the random geometric graph and the square graph for various distances.

Three statistics were taken for the graphs. The number of nodes are the number of sensors. The cost will increase for a higher value of this. The average degree is a good statistic for determining the robustness of the network. When a network has a high average degree, it will be less likely be influenced greatly by a failing node. The longest path statistic is important for the response time. For a lower value of this, a signal will take at most the longest path number of transmission to get to the last node.

From the table we see that the square graph performs better in terms of performance versus cost. When the square graph has the same number of nodes as the random graph, it outperforms it. From these statistics, the square graph with distance $d = 2.5$ is expected to perform well enough.

IV. Distributed Averaging

The objective is to calculate the average temperature of the sensors. This should be done without a centralized unit that collects all the values. This type of averaging is decentralized averaging. There are two major types of averaging. Synchronous averaging which will be discussed in Section IV-A. And decentralized averaging, which is discussed in Section IV-B.

A. Synchronous averaging

With synchronous averaging, all the sensors have a in sync clock. During each clock tick, the sensors will communicate with their neighbors and do some calculations. These can be seen an linear iterations. First define the model. Each node has a temperature value $x_i(k)$ where i notates node i and k iteration k. This can be written as the vector $\mathbf{x}(k) = [x_0(k) \quad x_1(k) \quad \dots \quad x_i(k) \quad \dots]^T$. To update this vector, the vector will be multiplied by the weight matrix W as shown in Eq. 1. The behaviour of W is picked such that Eq. 2 holds.

$$\mathbf{x}(k) = W\mathbf{x}(k-1) \tag{1}$$
$$\mathbf{x}(k) = W^k\mathbf{x}(0) \tag{2}$$

1) Optimal solution: Eq. 2 is giving some constraints on W such that an optimal matrix can be found. These conditions are found in Eqs. 3-6. ρ denotes the spectral radius of the matrix.

$$\lim_{k\to\infty} W^k = \frac{\mathbf{1}\mathbf{1}^T}{n} \tag{3}$$
$$\mathbf{1}^T W = \mathbf{1}^T \tag{4}$$
$$W\mathbf{1} = \mathbf{1} \tag{5}$$
$$\rho(W - \frac{\mathbf{1}\mathbf{1}^T}{n}) < 1 \tag{6}$$

These constraints and properties can be converted into a minimization problem shown in Eq. 7.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \rho(W - \frac{\mathbf{1}\mathbf{1}^T}{n}) \\ \text{subject to} \quad & W \in S, \quad \mathbf{1}^T W = \mathbf{1}^T, \quad W\mathbf{1} \quad = \mathbf{1} \end{aligned} \tag{7}$$

The problem with Eq. 7 is the complexity of the problem and the spectral function is not complex. Most complex problems are simple to solve with the help of gradient descends or other algorithms.

2) Optimal Symmetric solution: A solution to this problem would be to take an approximation of the spectral radius. If a matrix is symmetric, then the spectral radius coincides with the spectral norm[2]. This is shown in Eq. 8.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & ||W - \frac{\mathbf{1}\mathbf{1}^T}{n}||_2 \\ \text{subject to} \quad & W \in S, \quad W = W^T, \quad \mathbf{1}^T W = \mathbf{1}^T, \quad W\mathbf{1} \quad = \mathbf{1} \end{aligned} \tag{8}$$

The spectral norm is a convex function, which makes solving this problem substantially easier. However, this problem does have a scaling problem. For very big W, the problem becomes more complex.

3) Optimal constant weights solution: The simplify the shape of the W matrix, is setting all the edge weight equal to a constant. This gives the shape stated in Eq. 9. Where α is the constant weight and d_i is the degree of node i.

$$W_{ij} = \begin{cases} \alpha, & (i,j) \in E \\ 1 - \alpha d_i, & i = j \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

Finding the optimal α requires to express the spectral radius of $(W - \frac{\mathbf{1}\mathbf{1}^T}{n})$ into a function of the eigenvalues of the Laplacian of the adjacency matrix of the network. The optimal solution for alpha becomes then dependent on the biggest and the second smallest eigenvalue shown in Eq. 10.

$$\alpha^* = \frac{2}{\lambda_1(L) + \lambda_{n-1}(L)} \tag{10}$$

4) Maximum degree solution: A simple alternative to the previous problem is by choosing α to be dependent on the maximum degree of the network as defined in Eq 11. This will result in a worse performance, but convergence is guaranteed.

$$\alpha^* = \frac{1}{d_{max}} \tag{11}$$

5) Results: All the synchronous algorithms were tested on a downscaled problem since the full dimension problem would cost too much computing power. The dimension was downscaled to 21×12 meters. The results of the convergence of each algorithm is shown in Fig. 3. As expected, the performance decreases for every simplification of the algorithm.

A downside of the synchronous algorithm is, that the network has to be fully known in order to find the W matrix. Another downside is the dependability on a synchronized clock on all devices. The algorithms would also be very fragile for changes in the network.

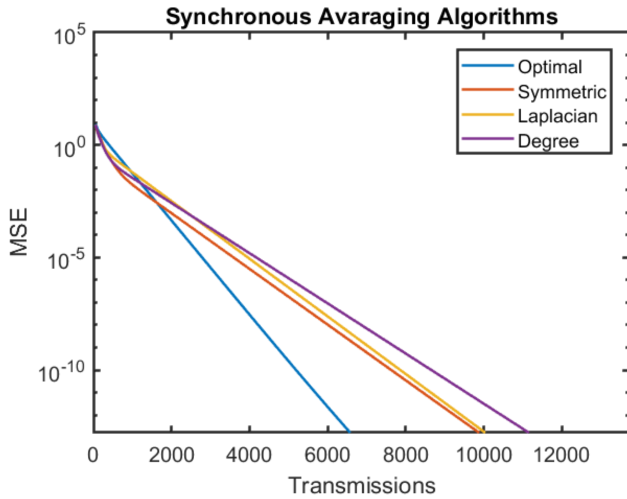

Fig. 3: Performance of the synchronous averaging algorithms.

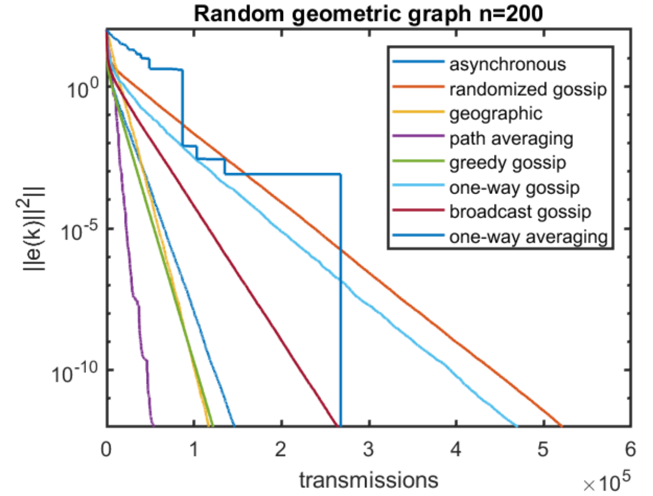

Fig. 4: Performance of the asynchronous averaging algorithms on a random graph.

B. Asynchronous averaging

To tackle most of the problems for the synchronous algorithm, a study on the asynchronous algorithms is done. Most of these algorithms require less calculations, and will this be explained stepwise for each iteration.

1) Distributed averaging: The distributed averaging works by picking random nodes and averaging locally. Each iteration requires 2 transmissions from the picked node and 1 transmission from each neighbor. Algorithm:

1) Select a node (i) uniformly at random.
2) Change the value of node i and all its neighbors to their shared mean.

2) Randomized gossip: This algorithm is the easiest of all and each iteration only requires 2 transmissions. Algorithm:

1) Select a node (i) uniformly at random.
2) Select a random node (j) and change the values of both nodes to their average.

3) Geographic gossip: This algorithm requires knowledge of all the node's location and has a variable number of transmissions needed. Algorithm:

1) Select a coordinate (x,y) and a node (i) at random.
2) Find the closest node to the coordinate (j).
3) Take the average of node i and j and change their values.
4) Number of transmissions: hop count from i to j times 2.

4) Randomized path averaging: This algorithm works the same as geographic gossip, but instead of updating two nodes, it will take the average of the whole path and change all the values of these nodes. This algorithm uses the same amount of transmissions per iteration as the geographic gossip.

5) Greedy gossip with eavesdropping: The transmission is supposed to be of broadcast type. Each neighbor keeps the values of its neighbors. A need of two transmissions is needed to inform the neighbors of the change. Algorithm:

1) Pick random node (i).
2) Choose the neighbor (j) with the most different value.
3) Take the average of node i and j and change their values.

6) One-Way gossip: For the next algorithms, two new matrices are defined. The weight ($\mathbf{w}(k)$) vector and the sum ($\mathbf{s}(k)$) vector. The average value is received by dividing $\mathbf{s}(k)/\mathbf{w}(k)$ pointwise. These vectors are updated iteratively by multiplying them with the matrix $D(K)$. This update function is shown in Eqs. 12 and 13. The easiest implementation is the One-Way gossip. The $D(k)$ matrix has the form shown in Eq. 15.

$$\mathbf{s}(k) = D(k)\mathbf{s}(k-1) \tag{12}$$
$$\mathbf{w}(k) = D(k)\mathbf{w}(k-1) \tag{13}$$
$$\tag{14}$$

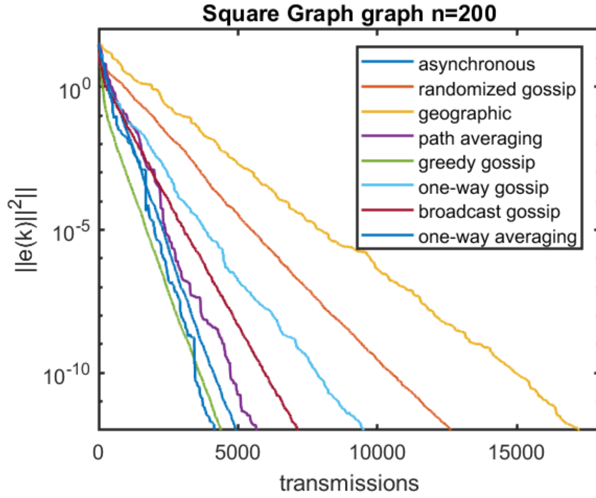

Fig. 5: Performance of the asynchronous averaging algorithms on the reference graph.

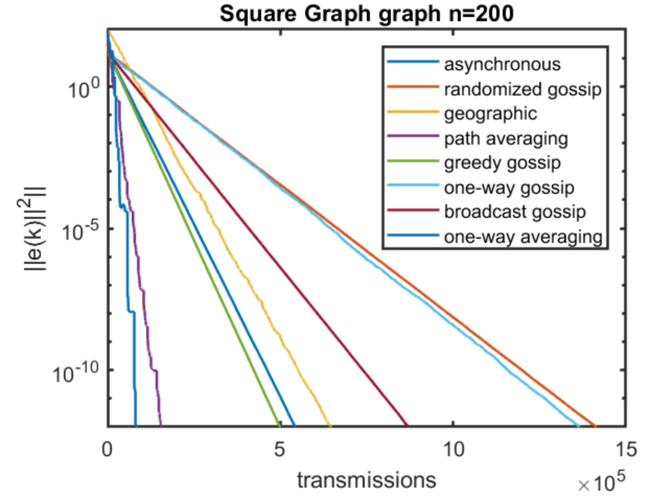

Fig. 6: Performance of the asynchronous averaging algorithms on the full scale graph.

$$D(k) = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & 1 \\ 0 & 0 & I)n-2 \end{bmatrix} \tag{15}$$

7) Broadcast weighted gossip: Broadcast weighted gossip picks a random node which then broadcasts its values. These values are then added to the neighbors. The D matrix will have the form shown in Eq. 16.

$$D_{i \to N(i)}(k) = \begin{bmatrix} \frac{1}{d_i+1} & 0 & \dots & 0 \\ \frac{1}{d_i+1} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ \frac{1}{d_i+1} & 0 & \dots & 1 \end{bmatrix} \tag{16}$$

8) One-Way averaging: The last algorithm is the On-Way averaging. It has the same structure as the geographic gossip, but the transmissions won't return. Each transmission to the coordinate will alter the values a slightly different way. The previous value and current value is important, together with a scaling value N. The D matrix has the form of equation 17.

$$D(k) = \begin{bmatrix} \frac{1}{N} & 0 & \dots & 0 & 0 & 0 \\ \frac{1}{N} & \frac{1}{N-1} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 & 0 \\ \frac{1}{N} & \frac{1}{N-1} & \dots & \frac{1}{2} & 0 & 0 \\ \frac{1}{N} & \frac{1}{N-1} & \dots & \frac{1}{2} & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & I_{n-N} \end{bmatrix} \tag{17}$$

9) Results: The results of each algorithm will be shown in a few figure. To compare the results with the synchronous methods, the downscaled version will be tested (Fig 5). To check if the algorithms are correctly implemented, a figure with the random graph will be shown (Fig 4). And lastly the figure with the full scale problem (Fig. 6). Each algorithms has its own downside and upsides. Often the decision for the algorithm depend on the specifications.

V. Fire Detection

The main objective of this report is the detection of fire. For this, objective based programming is used to model and simulate the fire. Two classes are introduces, the Sensor class and the Network class.

The Sensor class contains a few properties. The x and y coordinates, its temperature T, its memory value $value$, its state $state$ and some state memories $numD$, $numFA$ and $numFD$ which will be explained later on.

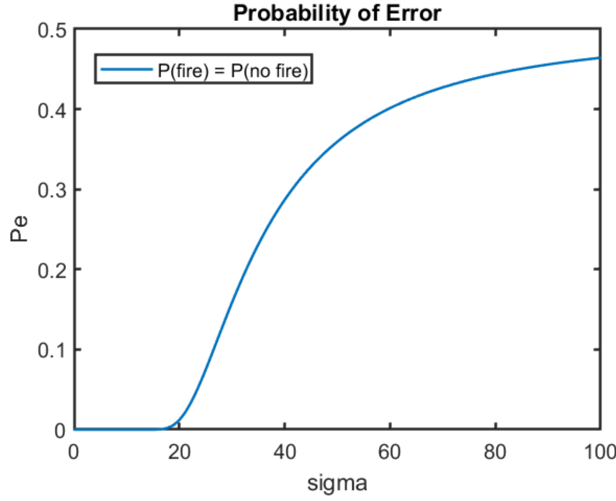

Fig. 7: Probability of false detection or misdetection as function of σ.

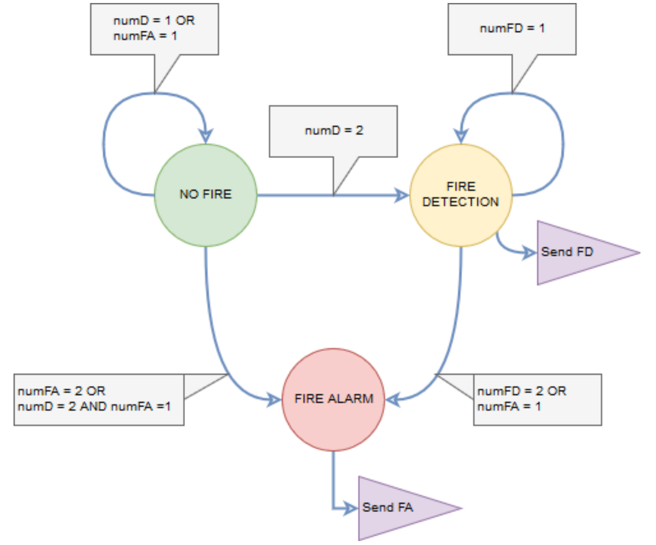

Fig. 8: State diagram of the Fire Detection system.

The Network class consists of 3 properties. The graph object G, the number of sensors n and an array of the Sensor objects $Sensors$.

The problem can be divided in a two parts. The fire detection and the communication about the fire.

A. Fire detection

The fire detection is fairly easy since the two distributions are far ahead. Since we have two Gaussian distribution, a good threshold value would be the average of the two means. This threshold value would then be $20 \times 80 \times 0.5 = 50°C$. If the sensor measures above the threshold, a fire is detected, otherwise the normal temperature is assumed. dependent on σ^2, an error could occur. The error for a false detection (P_{FD}) or a misdetection (P_{MD}) combined is defined by the Q-function. This function is shown in Eq 18 and depends on this σ. The function is also plotted in Fig. 7.

$$P_e = P_{FD} + P_{MD} = Q(\frac{(\mu_1 - \mu_0)^2}{4\sigma^2}) \tag{18}$$

For these two error, only one can be prevented. The false detection can be prevented by making the system robust. One sensor detection should not trigger the Fire Detection system but instead should wait for another detection. This solution is presented in the next Section.

B. Fire communication system

When a fire is detected, the system should quickly respond. But because this false alarm error could be possible, the system should also be robust. A robust system has been proposed by Khadivi and Hasler[3]. Their system proposed a FSM system where a sensor can has three different states: No Fire, Fire Detection and Fire Alarm. This this state diagram is shown in Fig. 8. In this system we have 3 signals: A signal when the sensor detects a fire, a signal when a neighbor sends a Fire Detection signal and a signal when a neighbor sends a Fire Alarm signal. On the hand of these variables, the sensor can change states.

To first test this system, we try it on different values for the radius (size) of the fire and the distance the sensors in the square network. These tests are shown in Fig. 9. Here wee see that the sensor distances larger than 2 do not reach a convergence in an all-out Fire Alarm state. Only from a distance of 2 meters do these states converge from approximately a fire with radius 2.5m.

We continue to test the performance. The next test is the convergence speed as a new function of different distances and radius. This is shown in Fig. 10. Here we see almost no difference in the convergence speed when the fire grows. Only that a minimum radius between the 2m and 3m is needed.

VI. Conclusion

This report has shown two different types of networks possible for the topology. It can be concluded that the square mesh network has a better performance and is more efficient per node. This is due to the unnecessary node close to each other in the random geometric graph.

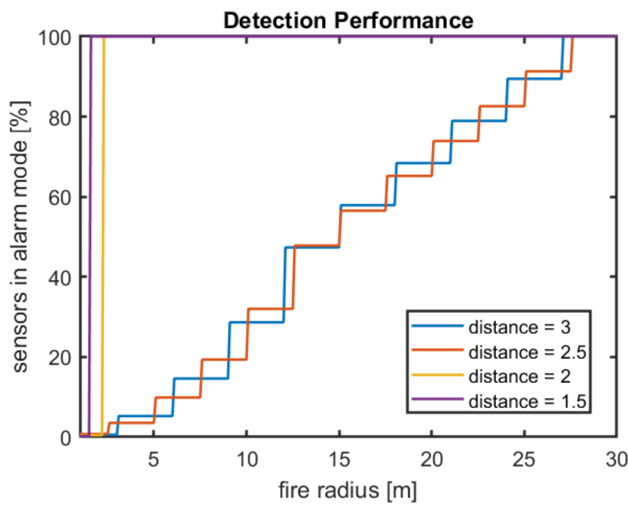

Fig. 9: Performance of the system for various fire radius and sensor distances.

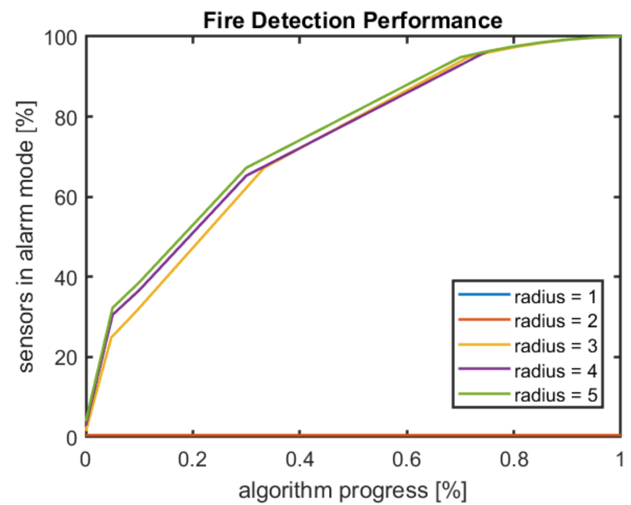

Fig. 10: Convergence Performance of the system for various fire radius and sensor distances.

After this we evaluated 12 different averaging algorithms for the system. Since a mesh network is widely known, this could be taken into account for the algorithms. However, it would depend on de robustness of the sensors. A synchronous network does not have the benefit since a synchronized clock would make the system too complex. If the sensors are robust then an algorithm as path averaging is the best option. If the sensors are not robust and can fail, a form of greedy gossip was our best option. We also noticed big difference in the type of topology where for example the geographic gossip performs very badly in a square mesh graph compared to the random graph.

The Fire detection system has shown to be robust and that it will converge for a sensor distance below 2 meters and a fire radius of around 2.5 meters.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011. [Online]. Available: http://dx.doi.org/10.1561/2200000016

[2] R. Heusdens, "Distributed signal processing," 2018, lecture slides and notes.

[3] A. Khadivi and M. Hasler, "Fire detection and localization using wireless sensor networks," in *SENSAPPEAL*, 2009.

Appendix A
MATLAB code

A. Scripts

averaging_async.m

```
1  clear
2  clc
3  close all
4
5  %% Variables
6  nodes = 200;              % num nodes for random graph
7  temperature = 20;         % average temperature
8  SNR = 9;                  % SNR on the sensors
9  transmissions = 8E5;      % number of transmissions
10 dimensions = [1 1];       % dimensions for random graph
11 dimensions = [21 12];     % dimensions for square graph
12 radius = 3;               % reach of sensor
13 distance = 3;             % placement between sensors (square graph)
14
15 %% Generate data and graph
16 rng(7)                                                          % prevent random
       problems
17 [G, c] = rnd_geograph(nodes, sqrt(log(nodes)/nodes), [1 1]);    % random graph
18 [G, c] = squaregraph(dimensions, radius, distance);             % square graph
19 T = awgn(temperature * ones(size(c,1), 1), SNR, 'measured');    % create random temp
       for sensors
20 xavg = mean(T)*ones(size(T));                                   % average vector
21
22 %% Average algorithms
23 disp('Asynchronous Averaging... 0/8');
24 [t1 err1] = asynchronous(G, T, xavg);
25 disp('Random Gossip... 1/8');
26 [t2 err2] = random_gossip(G, T, xavg);
27 disp('Geometric Gossip... 2/8');
28 [t3 err3] = geometric(G, T, xavg, c, dimensions);
29 disp('Randomized Path Averaging... 3/8');
30 [t4 err4] = path_averaging(G, T, xavg, c, dimensions);
31 disp('Greedy Gossip... 4/8');
32 [t5 err5] = greedy(G, T, xavg);
33 disp('One-Way Gossip... 5/8');
34 [t6 err6] = oneway_gossip(G, T, xavg);
35 disp('Broadcast Weighted Gossip... 6/8');
36 [t7 err7] = broadcast(G, T, xavg);
37 disp('One-Way Averaging... 7/8');
38 [t8 err8] = oneway_averaging(G, T, xavg, c, dimensions, 5);
39
40 %% plotting
41 figure;
42 semilogy(linspace(0, t1, length(err1)), err1); hold on
43 semilogy(linspace(0, t2, length(err2)), err2);
44 semilogy(linspace(0, t3, length(err3)), err3);
45 semilogy(linspace(0, t4, length(err4)), err4);
46 semilogy(linspace(0, t5, length(err5)), err5);
47 semilogy(linspace(0, t6, length(err6)), err6);
48 semilogy(linspace(0, t7, length(err7)), err7);
49 semilogy(linspace(0, t8, length(err8)), err8);
50 ylim([1E-12 1E2]);
51 title('Square Graph graph n=200');
52 xlabel('transmissions');
53 ylabel('||e(k)||^2||');
```

```
54 legend('asynchronous', 'randomized gossip', 'geographic', 'path averaging', 'greedy
     gossip', 'one-way gossip', 'broadcast gossip', 'one-way averaging');
```

---

averaging_sync.m

```
1  clear
2  clc
3  close all
4
5  %% Variables
6  dimensions = [21 12];    % room is 60 by 30
7  radius = 3;              % sensor radius is 3 m
8  dist = 3;                % sensors distance apart (squaregraph)
9  temperature = 20;        % average temperature
10 SNR = 10;                % SNR on the sensors
11
12 %% Generate data and graph
13 [G, c] = squaregraph(dimensions, radius, dist);      % square graph
14 T = awgn(temperature * ones(size(c,1), 1), SNR, 'measured');
15 average = mean(T)*ones(size(T));
16
17 %% Average algorithms
18 Ad = adjacency(G);
19
20 % minimization problem
21 fun = @optimal_minimization;
22 W0 = (full(Ad) + eye(size(Ad)))./4;
23 A = [];
24 b = [];
25 Aeq = [];
26 beq = [];
27 lb = zeros(size(Ad));
28 ub = full(Ad) + eye(size(Ad));
29 nonlcon = @optimal_equality;
30 options = optimoptions('fmincon', 'MaxFunctionEvaluations', 20000);
31
32 W_optimal = fmincon(fun,W0,A,b,Aeq,beq,lb,ub, nonlcon, options);
33
34 % symmetric minimization problem
35 fun = @symmetric_minimization;
36 nonlcon = @symmetric_equality;
37
38 W_symmetric = fmincon(fun,W0,A,b,Aeq,beq,lb,ub, nonlcon, options);
39
40 % optimal constant (Laplacian)
41 L = laplacian(G);
42 lambda = eig(L);
43 alpha = 2/(lambda(end) + lambda(2));
44
45 W_laplacian = alpha.* full(Ad) + diag(1-alpha*degree(G));
46
47 % max degree
48 alpha = 1/max(degree(G));
49
50 W_degree = alpha.* full(Ad) + diag(1-alpha*degree(G));
51
52 %% Calculate mse plot
53 for i = 1 : 500
54 mse_optimal(i) = mse(W_optimal^i*T, average);
55 mse_symmetric(i) = mse(W_symmetric^i*T, average);
56 mse_laplacian(i) = mse(W_laplacian^i*T, average);
```

```
57 mse_degree(i) = mse(W_degree^i*T, average);
58 end
59
60 %% plotting
61 tpi = sum(full(Ad(:)));
62 t = tpi * (1 : 500);
63
64 figure;
65 semilogy(t, mse_optimal); hold on;
66 semilogy(t, mse_symmetric);
67 semilogy(t, mse_laplacian);
68 semilogy(t, mse_degree);
69 title('Synchronous Avaraging Algorithms');
70 xlabel('Transmissions');
71 ylabel('MSE');
72 legend('Optimal', 'Symmetric', 'Laplacian', 'Degree');
73
74 %% Functions for optimization
75 function f = optimal_minimization(W)
76     f = eigs((W - ones(size(W))/size(W,1)), 1, 'lm');
77 end
78
79 function f = symmetric_minimization(W)
80     f = norm((W - ones(size(W))/size(W,1)));
81 end
82
83 function [c,ceq] = optimal_equality(W)
84     c = [];
85     o1 = W*ones(length(W),1) -  ones(length(W),1);
86     o2 = ones(1,length(W))*W - ones(1,length(W));
87     ceq = o1 + o2;
88 end
89
90 function [c,ceq] = symmetric_equality(W)
91     c = [];
92     o1 = W*ones(length(W),1) -  ones(length(W),1);
93     o2 = pinv(W) * W' - eye(size(W));
94     ceq = o1 + o2;
95 end
```

distributions.m

```
1 clear
2 clc
3 close all
4
5 %%
6 u0 = 20;
7 u1 = 80;
8 sigma = 0 : 0.1 : 100;
9
10 %% Get errors
11 Pe = qfunc((u1-u0).^2./(4*sigma.^2));
12 plot(sigma, Pe);
13 title('Probability of Error')
14 xlabel('sigma')
15 ylabel('Pe');
16 legend('P(fire) = P(no fire)');
```

fire_detection.m

```
clear
clc
close all

%% Variables
temperature = 20;        % average temperature
sigma = 10;                  % SNR on the sensors
dimensions = [60 30];    % dimensions for square graph
radius = 3;              % reach of sensor
distance = 2;            % placement between sensors (square graph)
% 1.5 for middle fire, 4.5 for corner fire

%% Create Network
[G, c] = squaregraph(dimensions, radius, distance);       % square graph
x = normrnd(temperature, sigma, size(c,1), 1);

%% Detection
detection = [];
for i = 1 : 5
Net = Network(G, c, x);                        % reset network states
Net.startFire([30 15], 3, 80, sigma);    % start fire
p = Net.fireDetection();                       % start detection algorithm
plot(linspace(0, 1, length(p)), p); hold on;     % plot
end

title('Fire Detection Performance');
xlabel('algorithm progress [%]');
ylabel('sensors in alarm mode [%]');
legend('radius = 1', 'radius = 2', 'radius = 3', 'radius = 4', 'radius = 5')
```

graph_geo_node_sim.m

```
clear
clc
close all

%% Variables
dimensions = [60 30];    % room is 60 by 30
radius = 3;              % sensor radius
range = 750;             % range of the nodes
it = 5;                  % #iterations for averaging

%% Start iterations
components = [];     % number of components per #nodes
tic;                 % measure time

% for all ranges
for nodes = 1 : range
    t = toc;         % measure time and display progress
    disp(['Progress: ' num2str(nodes/range * 100) '%' 9 'in: ' num2str(t) ' seconds'])
        ;
    tic;             % measure time
    comp = [];

    % iterate to average #nodes
    for i = 1 : it
        x = dimensions(1).*rand(nodes,1);
        y = dimensions(2).*rand(nodes,1);
        c = [x y];  % random coordinates

        % generate adjacenct matrix
```

```
29          A = dist(c, c');
30          A(A < 3) = 1;
31          A(A > 3) = 0;
32          A = A - eye(nodes);
33
34          % generate Graph Object and check #components
35          G = graph(A);
36          comp = [comp (conncomp(G))];
37      end
38      components = [components mean(comp)];
39  end
40
41  %% plotting
42  plot(1 : range, components)
43  title('Connected components')
44  xlabel('# sensors')
45  ylabel('# connected components');
46  axis tight
```

---

graphgeneration.m

```
1   clear
2   clc
3   close all
4
5   %% Variables
6   dimensions = [60 30];   % room is 60 by 30
7   radius = 3;             % sensor radius is 3 m
8   nodes = 400;            % number of sensors (geograph)
9   dist = 2.5;               % sensors distance apart (squaregraph)
10
11  %% Generate graphs
12  [G1, c1] = squaregraph(dimensions, radius, dist);     % square graph
13  [G2, c2] = rnd_geograph(nodes, radius, dimensions);   % random geo graph
14
15  %% Statistics
16  d1 = mean(degree(G1))
17  d2 = mean(degree(G2))
18  lp1 = max(max(distances(G1)))
19  lp2 = max(max(distances(G2)))
20  %% plotting
21  figure;
22  scatter(c1(:,1), c1(:,2), 'LineWidth', 1.5); hold on;
23  scatter(c2(:,1), c2(:,2));
24  title('Sensor placement');
25  xlabel('X-axis [m]');
26  ylabel('Y-axis [m]');
27  legend('Square graph', 'Random graph');
```

B. Functions

asynchronous.m

```
1   function [t err] = asynchronous(G, x, T, animate)
2   if nargin < 4
3       animate = false;
4   end
5
6   if animate
7      f = figure;
8   end
```

```
9
10 % variables
11 n = numnodes(G);
12 d = degree(G);
13 err = Inf;
14 k = 0;
15 t = 0;
16
17 % iterate
18 while (err(end) > 1E-12)
19     i = randi(n);            % choose random node
20     nei = neighbors(G,i);    % get neighbors
21
22     x([i ;nei]) = sum(x([i ;nei]))./(d(i)+1);
23     t = t + 2 + d(i);
24
25     k = k + 1;
26     err(k) = norm(x - T);
27
28     if animate && (mod(k,100) == 0)
29         semilogy(err);
30         drawnow
31     end
32 end
33 if animate
34     close(f)
35 end
36 end
```

broadcast.m

```
1 function [t, err] = broadcast(G, x, T, animate)
2 if nargin < 4
3     animate = false;
4 end
5
6 if animate
7     f = figure;
8 end
9
10 % variables
11 n = numnodes(G);
12 err = Inf;
13 k = 0;
14 t = 0;
15 s = x;
16 w = ones(n, 1);
17 % iterate
18 while (err(end) > 1E-12)
19     i = randi(n);
20     nei = neighbors(G,i);
21
22     D = eye(length(nei) + 1);
23     D(:,1) = 1/(length(nei) +1);
24
25     s([i ;nei]) = D * s([i ;nei]);
26     w([i ;nei]) = D * w([i ;nei]);
27
28
29     x = s./w;
30
```

```
31      k = k + 1;
32      err(k) = norm(x - T);
33
34      if animate && (mod(k,100) == 0)
35          semilogy(err);
36          drawnow
37      end
38  end
39  t = k;
40  if animate
41      close(f)
42  end
43
44  end
```

---

geometric.m

```
1   function [t, err] = geometric(G, x, T, c, dimension, animate)
2   if nargin < 6
3       animate = false;
4   end
5
6   if animate
7       f = figure;
8   end
9
10  % variables
11  n = numnodes(G);
12  err = Inf;
13  k = 0;
14  t = 0;
15
16  % iterate
17  while (err(end) > 1E-12)
18      i = randi(n);             % choose random node
19      cr = dimension.*rand(1,2);
20      dis = dist(c, cr');
21      j = find(dis == min(dis));
22      [P, hops] = shortestpath(G, i, j);
23
24      x([i j]) = (x(i)+x(j))/2;
25      t = t + 2*hops;
26
27      k = k + 1;
28      err(k) = norm(x-T);
29
30      if animate && (mod(k,100) == 0)
31          semilogy(err);
32          drawnow
33      end
34  end
35
36  if animate
37      close(f)
38  end
39
40  end
```

---

greedy.m

```
1   function [t, err] = greedy(G, x, T, animate)
```

```
if nargin < 4
    animate = false;
end

if animate
    f = figure;
end

% variables
j = numnodes(G);
err = Inf;
k = 0;
t = 0;

% iterate
while (err(end) > 1E-12)
    i = randi(j);           % choose random node
    nei = neighbors(G,i);
    diff = dist(x(nei), x(i)');
    n = nei(find(diff == max(diff), 1));

    x([i n]) = (x(i) + x(n))/2;
    t = t + 2;

    k = k + 1;
    err(k) = norm(x-T);

    if animate && (mod(k,100) == 0)
        semilogy(err);
        drawnow
    end
end

if animate
    close(f)
end

end
```

---

oneway_averaging.m

```
function [t, err] = oneway_averaging(G, x, T, c, dimension, maxN, animate)
if nargin < 7
    animate = false;
end

if animate
    f = figure;
end

% variables
n = numnodes(G);
err = Inf;
k = 0;
t = 0;
s = x;
w = ones(n, 1);
d = degree(G);

% iterate
while (err(end) > 1E-12)
```

```
21      i = randi(n);
22      cr = dimension.* rand(1,2);     % random location
23      distances = dist(c, cr');        % find nearest node to location
24      j = find(distances == min(distances));
25      [P,hops] = shortestpath(G, i, j);  % calc #hops
26
27      N = length(P);
28      D = tril(kron(fliplr(1:N).^-1, ones(N,1)));
29
30      if N > maxN
31          D = tril(kron(fliplr(1:maxN).^-1, ones(maxN,1)));
32          P = P(1:maxN);
33      end
34
35      s(P) = D * s(P);
36      w(P) = D * w(P);
37
38      x = s./w;
39      t = t + d(i)+1;
40
41      k = k + 1;
42      err(k) = norm(x - T);
43
44      if animate && (mod(k,100) == 0)
45          semilogy(err);
46          drawnow
47      end
48  end
49
50  if animate
51      close(f)
52  end
53
54  end
```

oneway_gossip.m

```
1   function [t, err] = oneway_gossip(G, x, T, animate)
2   if nargin < 4
3       animate = false;
4   end
5
6   if animate
7       f = figure;
8   end
9
10  % variables
11  n = numnodes(G);
12  err = Inf;
13  k = 0;
14  t = 0;
15
16  s = x;
17  w = ones(n, 1);
18  D = [0.5 0; 0.5 1];
19
20  % iterate
21  while (err(end) > 1E-12)
22      i = randi(n);
23      nei = neighbors(G,i);
24      j = nei(randi(length(nei)));
```

```
    s([i j]) = D * s([i j]);
    w([i j]) = D * w([i j]);
    x([i j]) = s([i j])./w([i j]);

    k = k + 1;
    err(k) = norm(x - T);

    if animate && (mod(k,100) == 0)
        semilogy(err);
        drawnow
    end
end
t = k;
if animate
    close(f)
end

end
```

---

path_averaging.m

```
function [t, err] = path_averaging(G, x, T, c, dimension, animate)
if nargin < 6
    animate = false;
end

if animate
    f = figure;
end

% variables
n = numnodes(G);
err = Inf;
k = 0;
t = 0;

% iterate
while (err(end) > 1E-12)
    i = randi(n);           % choose random node
    cr = dimension.*rand(1,2);
    dis = dist(c, cr');
    j = find(dis == min(dis));
    [P, hops] = shortestpath(G, i, j);

    x(P) = sum(x(P))/length(P);
    t = t + 2*hops;

    k = k + 1;
    err(k) = norm(x-T);

    if animate && (mod(k,100) == 0)
        semilogy(err);
        drawnow
    end
end

if animate
    close(f)
end

```

```
end
```

random_gossip.m

```
function [t,err] = random_gossip(G,x, T, animate)
if nargin < 4
    animate = false;
end

if animate
    f = figure;
end

% variables
n = numnodes(G);
d = degree(G);
err = Inf;
k = 0;


% iterate
while (err(end) > 1E-12)
    i = randi(n);            % choose random node
    nei = neighbors(G,i);    % get neighbors
    j = nei(randi(d(i)));

    x([i j]) = (x(i)+x(j))/2;

    k = k + 1;
    err(k) = norm(x - T);

    if animate && (mod(k,100) == 0)
        semilogy(err);
        drawnow
    end
end
t = k * 2;
if animate
    close(f)
end
end
```

rnd_geograph.m

```
function [G, c] = rnd_geograph(nodes,radius, dimensions)
    components = 2;     % #components
    overflow = 1;       % prevent loop of death

    while components > 1    % while not all sensors are connected
        disp(['Try number: ' num2str(overflow)]);   % display progress
        overflow = overflow + 1;
        if overflow > 1000  % no more than 1000 tries allows
            error('Did not manage to creat graph');
        end

        % generate random coordinates
        x = dimensions(1).*rand(nodes,1);
        y = dimensions(2).*rand(nodes,1);
        c = [x y];  % coordinates

        % create adjacency matrix
```

```
18          A = dist(c, c');
19          A(A < radius) = 1;
20          A(A > radius & A ~= 1) = 0;
21          A = A - eye(nodes);
22
23          % generate Graph Object and count #components
24          G = graph(A);
25          components = max(conncomp(G));
26      end
27  end
```

---

squaregraph.m

```
1  function [G c] = squaregraph(dimensions, radius, distance)
2  % standard coordinate values
3  x = distance : distance : dimensions(1) - distance;
4  y = fliplr(distance : distance : dimensions(2) - distance)';
5
6  % create coordinates
7  X = repmat(x, [length(y) 1]);
8  Y = repmat(y, [1 length(x)]);
9  c = [X(:) Y(:)];
10
11 % create graph
12 A = dist(c, c');
13 A(A < radius) = 1;
14 A(A > radius) = 0;
15 A = A - eye(length(A));
16 G = graph(A);
17 end
```

C. Classes

Network.m

```
1  classdef Network
2      properties
3          G;          % graph
4          n;          % number of sensors
5          Sensors;    % Sensors array
6      end
7
8      methods
9
10         % CONSTRUCTOR
11         function obj = Network(G, c, T)
12             obj.G = G;
13             obj.n = numnodes(G);
14             for i = 1 : obj.n
15                 S(i) = Sensor(c(i,1), c(i,2), T(i));
16             end
17             obj.Sensors = S;
18         end
19
20         % START FIRE AT LOCATION, WITH RADIUS AND TEMP
21         function startFire(obj, location, radius, temperature, sigma)
22             c = [[obj.Sensors.x]', [obj.Sensors.y]'];
23             distance = dist(c, location');
24             idx = find(distance < radius);
25             for i = 1 : length(idx)
26                 obj.Sensors(idx(i)).T = normrnd(temperature, sigma);
```

```
27              end
28          end
29
30          % AVERAGE THE TEMP WITH ASYNCHRONOUS AVERAGING
31          function asyncAverage(obj, transmissions)
32              tpi = 2 + mean(degree(obj.G));  % average transmissions per iteration
33              for i = 1 : fix(transmissions/tpi)
34                  node = randi(obj.n);
35                  nei = neighbors(obj.G,node);
36
37                  obj.Sensors(node).value = sum([obj.Sensors([node;nei]).value])/(length(
                        nei)+1);
38                  for j = 1 : length(nei)
39                      obj.Sensors(nei(j)).value = obj.Sensors(node).value;
40                  end
41              end
42          end
43
44          % GET AVERAGE TEMP
45          function temp = getAverageTemp(obj)
46              temp = mean([obj.Sensors.T]);
47          end
48
49          % GET MSE
50          function mse = getMSE(obj)
51              temp = obj.getAverageTemp();
52              mse = norm([obj.Sensors.value] - temp);
53          end
54
55          % GIVES WHICH SENSORS DETECT FIRE
56          function idx = detectsFire(obj)
57              idx = [];
58              for i = 1 : obj.n
59                  if (obj.Sensors(i).detectsFire()) || (obj.Sensors(i).state > 0)
60                      idx = [idx i];
61                  end
62              end
63          end
64
65          % FIRE DETECTION ALGORITHM
66          function p = fireDetection(obj)
67              % let sensors first detect the fire once
68              for i = 1 : obj.n
69                 obj.Sensors(i).detectFire();
70              end
71
72              % progress vector
73
74
75              % loop every sensor until nothing changes anymore
76              p = [];
77              changed = true;
78              while changed
79                  p = [p 100*length(obj.detectsFire())/obj.n];
80                  changed = false;
81                  for i = 1 : obj.n   % loop every sensor
82                      % update states and detect for fires (in child function)
83                      [tf, sendsignal] = obj.Sensors(i).checkState();
84                      changed = or(changed, tf);  % if changed let iterate again
85
86                      % send messages if state has changed
87                      if sendsignal == 1
```

```
88                          nei = neighbors(obj.G, i);
89                          for j = 1 : length(nei)
90                              obj.Sensors(nei(j)).numFD = obj.Sensors(nei(j)).numFD + 1;
91                          end
92                      end
93                      if sendsignal == 2
94                          nei = neighbors(obj.G, i);
95                          for j = 1 : length(nei)
96                              obj.Sensors(nei(j)).numFA = obj.Sensors(nei(j)).numFA + 1;
97                          end
98                      end
99                  end
100             end
101         end
102
103     end
104
105 end
```

Sensor.m

```
1  classdef Sensor < handle
2      %   Sensor is a class for the fire detecting sensor
3      %   Has 3 states, IDLE = 0, ER = 1 (ERROR RECIEVED) and ED = 2 (ERROR DETECTED)
4      %   Has coordinates of its placement
5      %   Has a measures temperature T
6
7      properties
8          x; y;                   % x and y positions
9          T;                      % temperature
10         value;                  % value stored as temperature
11         state;                  % state
12         numD; numFA; numFD;     % variables for state
13     end
14
15     methods
16         % CONSTRUCTOR
17         function obj = Sensor(x, y, T)
18             obj.x = x; obj.y = y;
19             obj.T = T; obj.value = T;
20             obj.state = 0; obj.numD = 0;
21             obj.numFA = 0; obj.numFD = 0;
22         end
23
24         % CHECKS IF IT DETECTS FIRE
25         function tf = detectsFire(obj)
26             tf = obj.T > 50;
27         end
28
29         % ADDS numD if it detects fire
30         function detectFire(obj)
31             % detect fire then
32             if obj.detectsFire()
33                 obj.numD = obj.numD + 1;
34             end
35         end
36
37         % CHECK AND CHANGE STATE, IF STATE CHANGE, SEND SIGNAL TO NEIGHBORS
38         function [tf, sendsignal] = checkState(obj)
39             obj.detectFire();
40             oldstate = obj.state;
```

```
41              if (obj.numD < 2) && (obj.numFA < 2)
42                  obj.state = 0;
43              elseif (obj.numD > 1) && (obj.numFA > 0)
44                  obj.state = 2;
45              elseif (obj.numFA > 1)
46                  obj.state = 2;
47              elseif (obj.numD > 1) && (obj.numFD > 1)
48                  obj.state = 2;
49              else
50                  obj.state = 1;
51              end
52              tf = obj.state ~= oldstate;
53
54              if tf && (obj.state == 1)
55                  sendsignal = 1;
56              elseif tf && (obj.state == 2)
57                  sendsignal = 2;
58              else
59                  sendsignal = 0;
60              end
61          end
62
63      end
64  end
```