Homework #5

AA 597: Networked Dynamics Systems
Prof. Mehran Mesbahi
Due: Feb 16, 2024 11:59pm
Soowhan Yi

All the codes are available at the end of the documents or here. https://github.com/SoowhanYi94/ME597

P1. 7.1 This chapter mainly dealt with Δ-disk graphs, that is, proximity graph (V,E) such that $v_i, v_j \in E$ if and only if $||x_i - x_j|| \leq \Delta$. wjere $x_i \in R^p, i = 1, \cdots, n$, is the state of robot i. In this exercise, we will be exploring another type of proximity graphy, namely the wedge graph. Assume that instead of single integrator dynamics, the agents' dynamics are defined as unicycle robots, that is,

$$\dot{x}_i(t) = v_i(t)cos\phi_i(t)$$
$$\dot{y}_i(t) = v_i(t)sin\phi_i(t)$$
$$\dot{\phi}_i(t) = \omega_i(t)$$

Here $[x_i, y_i]^T$ is the position of robot i, while, ϕ_i denotes its orientation. Moreover, v_i and ω_i are the translational and rotational velocities, which are the controlled inputs. Now, assume that such a robot is equiopped with a rigidly mounted camer, facing in the forward direction. This gives rise to a directed wedge graph, as seen in the figure. For such a setup, if robot j is visible from robot i, the available information is $d_{ij} = ||[x_i, y_i]^T - [x_j, y_j]^T||$ (distance between agents) and $\delta\phi_{ij}$(relative interagent angle) as per the figure below. Explain how you would solve the rendezvous (agreement) problem for such a system.

In order to solve this agreement problem, we need to controll the distance between two agents, especially between the leaf nodes(v_j and v_k in the example) and the center node(v_i), and angle between them. We have a sector form of sensing area, and we need to keep them inside in order to stay connected. So we construct a potential function of $V_i = \sum_{j\in N(i)}(\frac{1}{(\delta\phi_{ij}-\frac{\Delta\psi}{2})^2} + \frac{d_{ij}^2}{\rho - d_{ij}})$. Then,

$$v_i(t) = \dot{r}_i(t) = \sum_{j\in N(i)} -\frac{dV_i}{d(d_{ij})} = -\sum_{j\in N(i)} \frac{2\rho - ||d_{ij}||}{(\rho - ||d_{ij}||)^2}(r_i - r_j)$$
$$\dot{x}_i(t) = -\sum_{j\in N(i)} \frac{2\rho - ||d_{ij}||}{(\rho - ||d_{ij}||)^2}(r_i - r_j)\cos(\delta\phi_{ij})$$
$$\dot{y}_i(t) = -\sum_{j\in N(i)} \frac{2\rho - ||d_{ij}||}{(\rho - ||d_{ij}||)^2}(r_i - r_j)\sin(\delta\phi_{ij})$$

where this i corresponds to those robot agents. Lemma 7.1 from textbook proves that $\Omega(\rho, r_0) = \{r|V(\rho, r) \leq V(\rho, r_0)\}$ is invariant set under the above control law, where $r_0 \in D^{\epsilon}_{G,\rho}$ for $\epsilon \in (0, \rho)$. Also the theorem 7.2 from textbook proves that this control law asymptotically converges to the static centroid.
Now we need to keep the angles between them to be in certain range, and I would use the same logic that was used in above equation.

$$\omega_i = \delta\dot{\phi}_{ij}(t) = -\sum_{j\in N(i)} \frac{-2}{(\delta\phi_{ij} - \frac{\Delta\psi}{2})^3}$$

With above equations, agents are guranteed to stay in the sector shaped sensing area. Since we have $\delta\phi^0_{ij} \leq \frac{\Delta\psi}{2}$, this $\delta\phi_{ij}(t)$ would never increase. Therefore angle between them would never increase and distance would converge to their static centroid.

P2. Show that if a δ-disk proximity graph with 4 agents starts connected, then, using the linear agreement protocol, it stays connected for all times.
With theorem 3.4 in the textbook, we know that the linear agreement protocol converges to the agreement set. Therefore the length of edges between two vertices would never increase. Even with the case where 2 nodes (node 1 and node 2)are located in one points Δ away from node 3, and node 4 is located Δ away from node 3 and 2Δ away from node 1 and 2, we proved that they do converge in one point with 4 agents from last homework(4.8). As we have started with each lengths of edges being less than or equal to Δ, those lengths would stay less than Δ in the progress of the linear agreement protocol and until it reaches the agreement.

P3.

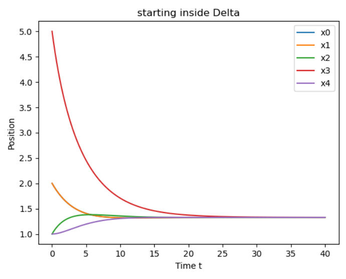
$$\dot{x}_i(t) = -\sum_{j \in N_{G_d}(i)} \frac{2(\Delta - ||d_{ij}||) - ||l_{ij} - d_{ij}||}{(\Delta - ||d_{ij}|| - ||l_{ij} - d_{ij}||)^2}(x_i(t) - x_j(t) - d_{ij})$$

Lemma 7.5
Given an initial conditions x_0 such that $y_0 = (x_0 - \tau_0) \in D^{\epsilon}_{G_d,\Delta-||d||}$, with G_d a connected spanning graph of $G(x_0)$, the group of autonomous mobile agents adopting the decentralized contral law (the equation above) are guranteed to satisfy

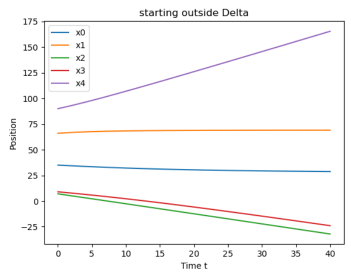
$$||x_i(t) - x_j(t)|| = ||l_{ij}(t)|| \le \Delta \text{ for all } t > 0 \text{ and } v_i, v_j \in E_d$$

Theorem 7.6 Under the same assumptions as in Lemma 7.5, for all i, j, the pairwize relative distants $||l_{ij}|| = ||x_i(t) - x_j(t)||$ asymptotically converge to $||d_{ij}||$ for $v_i, v_j \in E_d$

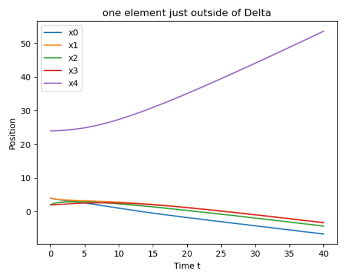
Implement the algorithm above and explore how the conditions of Lemma 7.5/Thm 7.6 are required for the algorithm to work as proposed.



(a) $||x_i - x_j|| \le \Delta$



(b) $||x_i - x_j|| \nleq \Delta$



(c) $||x_4 - x_j|| \nleq \Delta$

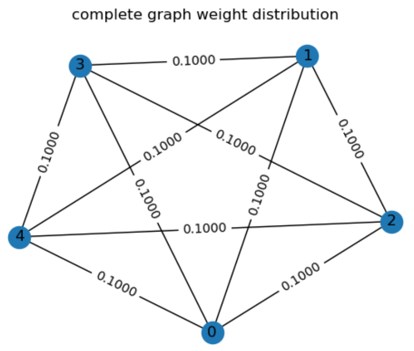
For above graphs, the desired lengths between each nodes are set to be 0 for simplicity. As we can see from above results, when those initial conditions satisfy those conditions of Lemma 7.5 and theorem 7.6, above algorithm asymptotically converges to edge length of 0. Also, those that are not satisfing those conditions (x_0, x_3, x_4 in (b), and x_4 in (c)) are diverging and those that are satisfing conditions (x_1, x_2 in (b), and x_0, x_1, x_2, x_3 in(c)) are converging. Also, when $||x_i - x_j|| = \Delta$, above algorithm does not compute because we would be getting infinite number when the desired length for edges are 0. So we need this threshold ϵ for computation. Therefore this algorithm, in order to work properly, those conditions of Lemma 7.5 and theorem 7.6 are required.

P4. 11.2 Show that if $\lambda_2(G)$ has an algebraic multiplicity m in in the graph G, then adding up to m edges will not improve it.
If $\lambda_2(G)$ has an algebraic multiplicity of m, there are m linearly independant eigenvectors assosicated with this $\lambda_2(G)$. Also the characteristic equation for this eigenvalues is

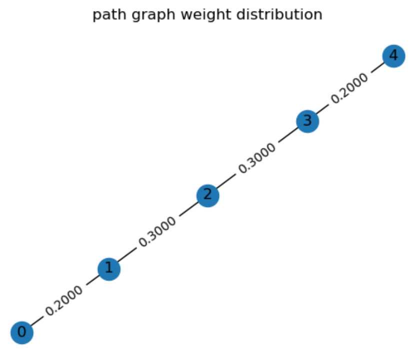
$$f(\lambda) = (\lambda_1 - \lambda)(\lambda_2 - \lambda) \cdots (\lambda_{m+1} - \lambda) \cdots (\lambda_n - \lambda) = 0$$

where $\lambda_2 = \lambda_3 = \cdots = \lambda_{m+1}$. If single edge is added, the connectivity increases, and increases the value of λ_2. However, the eigenvectors assosicated with those λs are linearly independant, and therefore the value of $\lambda_3, \lambda_4 \cdots \lambda_m$ remains the same, making the second smallest eigenvalue to become $\lambda_3, \lambda_4 \cdots \lambda_{m+1}$. Therefore adding edges upto m edges would not change the value of second smallest eigenvalue.

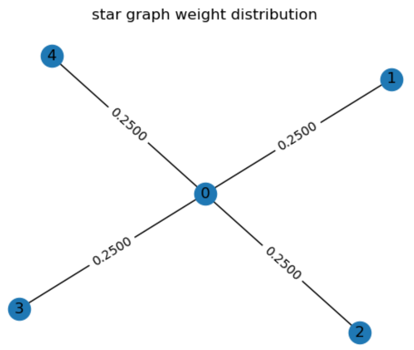
P5. 11.10 Use the approach of section11.6 and a semidefinite programming solver (such as the one mentioned in notes and references), to maximize $\lambda_2(G)$ for the weighted versions of K5 , P5 , and S5 , subject to a normalization on the sum of the weights. Comment on any observed patterns for the optimal weight distribution.



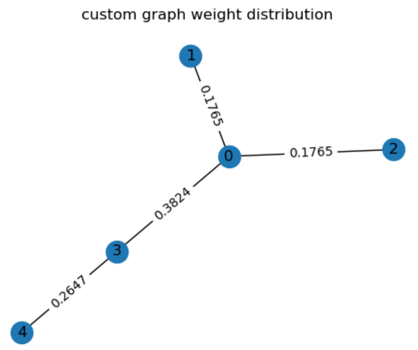
(a) K_5 weight distribution



(b) P_5 weight distribution



(c) S_5 weight distribution



(d) Textbook example

We see that those weight are summed to 1 for all graphs and they are symmetric respect to their edges, in case of K_5, P_5, S_5. Of course the custom graph(textbook example) would not have symmetric weights for edges that are not symmetric. But the edge connecting (0,1) and (0,2) are symmetric and their weights are the same.

Code P3

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

def show(graph):
    plt.figure()
    poses = nx.get_node_attributes(graph, 'pos')
    nx.draw_networkx_edges(graph, pos = poses,edgelist=graph.edges(),arrows=True)
    nx.draw_networkx_nodes(graph, pos = poses, nodelist=graph.nodes() ,label=True)
    nx.draw_networkx_labels(graph, pos=poses)
    plt.show()

def random_graphs_init(graph,num, Delta):
    poses = {i: np.random.randint(0,Delta) for i in range(num)} # starts inside Delta
    # poses = {i: np.random.randint(0,10*Delta) for i in range(num)} # starts outside of Delta
    # poses_= []
    # for i in range(num):
    #     if i == num -1:
    #         poses_.append(np.max(poses_) + Delta +10)
    #     else :
    #         poses_.append(np.random.randint(0,Delta))
    # poses = {i: poses_[i] for i in range(num)} # one element just outside of Delta
    nx.set_node_attributes(graph,poses, "pos")
    edges = {edge: np.abs(graph.nodes[edge[0]]["pos"]-graph.nodes[edge[1]]["pos"])  for edge in graph.edg
    desired_l = 0
    nx.set_edge_attributes(graph, edges, "edge_length")

    return graph, desired_l
num = 5
k = 1
Delta = 10
labels = []
D = nx.gnm_random_graph(num, (num -1)*(num-2)/2, directed=False)
D,desired_l = random_graphs_init(D,num, Delta)
D = nx.minimum_spanning_tree(D)

def xdot(x, t,desired_l):
    poses = {i: x[i] for i in range(num)}
    nx.set_node_attributes(D, poses, "pos")
    edges = {edge: np.abs(D.nodes[edge[1]]["pos"]-D.nodes[edge[0]]["pos"])  for edge in D.edges()}
    nx.set_edge_attributes(D, edges, "edge_length")
    ret  = []
    for i in range(num):
        dotx = 0
        for j in nx.neighbors(D, i):
            if ((i,j) in edges) :
                numerator = 2*(Delta - np.abs(desired_l )) - np.abs(edges[(i,j)] - desired_l )
                denominator = (Delta - np.abs(desired_l ) - np.abs(edges[(i,j)] - desired_l ))**2
                rest = D.nodes[i]["pos"] - D.nodes[j]["pos"] - desired_l
                dotx += numerator/denominator *rest
            elif ((j,i) in edges):
                numerator = 2*(Delta - np.abs(desired_l )) - np.abs(edges[(j,i)] - desired_l )
                denominator = (Delta - np.abs(desired_l ) - np.abs(edges[(j,i)] - desired_l ))**2
                rest = D.nodes[i]["pos"] - D.nodes[j]["pos"] - desired_l
                dotx += numerator/denominator *rest
        ret.append(-dotx)

    return ret

def main():

    for i in range(num):
        labels.append(f"x{i}")
    t = np.linspace(0, 40,num= 1001)

    l=[]
    [l.extend([v]) for k,v in nx.get_node_attributes(D, 'pos').items()]

    trajectory_x = odeint(xdot, l, t, args=(desired_l, ))
    plt.figure()
    plt.plot(t, trajectory_x, label = labels)

    plt.xlabel("Time t")
    plt.ylabel("Position")
    plt.title("one element just outside of Delta")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```

Code P5

```
import cvxpy as cp
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
#https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process
def gramschmidt(V):
    n, m = V.shape
    U = np.zeros_like(V)
    U[:, 0] = V[:, 0] / np.linalg.norm(V[:, 0])
    for i in range(1, m):
        U[:, i] = V[:, i]
        for j in range(i):
            U[:, i] -= np.dot(U[:, j].T, U[:, i]) * U[:, j]
        U[:, i] /= np.linalg.norm(U[:, i])
    return U[:,1:]

def main():
    nums = [5]
    custom_graph = nx.star_graph(nums[0]-2)
    custom_graph.add_node(nums[0] -1)
    custom_graph.add_edge(nums[0]-1, nums[0]-2)
    for num in nums:
        names = ['complete', 'path', 'star', 'custom' ]
        # graphs = [ custom_graph]
        graphs = [ nx.complete_graph(num), nx.path_graph(num), nx.star_graph(num - 1),custom_graph]
        k =0
        for graph in graphs:
            U = gramschmidt(np.column_stack([np.ones(num), np.random.randn(num, num-1)]))
            x = cp.Variable((len(graph.edges),len(graph.edges)), diag=True)
            D_D = nx.incidence_matrix(graph, graph.nodes(),oriented=True).toarray()
            gamma = cp.Variable(1)
            objective = cp.Maximize(gamma)
            constraints = [x >>0, cp.trace(x) == 1,(U.T@D_D @ x @ D_D.T@U )>>gamma*np.eye(num-1)]
            prob = cp.Problem(objective, constraints)
            result = prob.solve()
            plt.figure()
            plt.title(f"{names[k]} graph weight distribution")
            edges = list(graph.edges())
            edge_labels = [ '%.4f' % elem for elem in x.value.data[0] ]
            weight = {edges[i]: edge_labels[i] for i in range(len(edges))}
            pos = nx.spring_layout(graph)
            nx.draw(graph,pos = pos, with_labels = True)
            nx.draw_networkx_edge_labels(graph, pos = pos, edge_labels=weight)
            k +=1
        plt.show()

if __name__ == "__main__":
    main()
```