M345SC 2019 Homework 2 Solutions

Due date/time: 1/3/2019, 23:59 GMT

Part 1. (10 pts)

For each question in this part, you should develop a correct, efficient algorithm, implement the algorithm in Python, and discuss the running time and efficiency of your implementation. Your discussion should include an estimate of the asymptotic running time and a concise description of how you constructed this estimate. You may use *only* numpy and the collections module in your final submission for part 1. No other external modules (networkx, scipy, etc...) are allowed in your codes for this part.

1) You are designing an automated assembly process for a revolutionary new smartphone. The process consists of N tasks. M of these N tasks require at least one other task to have been completed before the task itself can be started. Each task requires one day to complete (and must be started and finished within a day). You have an unlimited number of multitalented robots at your disposal which can each complete one task per day. What is the shortest number of days needed to assemble the phone?

You are provided a *dependency list*, L, as input. This is a list containing N sublists. The sublist, $L[i]$ contains the indices of tasks that must be completed before task i can be started ($N-M$ of these sublists will be empty). Complete the function, *scheduler* in *p1_dev.py* so that it efficiently assigns a day to each task so that the total number of days to complete all tasks is minimized. Tasks are numbered from *0* to *N-1* and the function should return an N-element list whose *ith* element is an integer corresponding to the day on which task i is completed. For example, if *M=0*, then you should return a list with N zeros. You may assume that there are no pairs of tasks where each requires the other to have been previously completed and that $N > M$. Add a discussion of your implementation to the docstring of your function.

Ans: The solution code is below:

```
#----------Solution function--------------------#
def scheduler(L):
    """
    Question 1.1
    Schedule tasks using dependency list provided as input

    Input:
    L: Dependency list for tasks. L contains N sub-lists, and L[i] is a sub-list
    containing integers (the sub-list my also be empty). An integer, j, in this
    sub-list indicates that task j must be completed before task i can be started.

    Output:
    S: A list of integers corresponding to the schedule of tasks. S[i] indicates
    the day on which task i should be carried out. Days are numbered starting
    from 0.

    Discussion: Add analysis here
    """

    #Problem setup
    N = len(L)
    queue = deque([])
    a = [[] for i in range(N)]
    Nd = np.zeros(N,dtype=int)

    #Loop through dependency list
    #if len(dependencies)=0 append to queue
    #create list of number of dependencies
    #create adjacency list a[i] = dependents of [i]
```

```
        S = [0 for i in range(N)]
        for i,l in enumerate(L):
            Nd[i] = len(l)
            if len(l)==0: queue.append(i)
            for n in l:
                a[n].append(i)

        #pop from queue, reduce number of dependencies for neighbors,
        #if num=0, append to queue
        #terminate when queue is empty
        while len(queue)>0:
            n = queue.popleft()
            for j in a[n]:
                Nd[j] = Nd[j]-1
                if Nd[j]==0:
                    S[j]=S[n]+1
                    queue.append(j)

        return S
    #---------------------------------------------#
```

The basic approach is to first (on day 0) carry out all tasks with zero dependencies, then on day 1, all tasks only dependent on day 0 tasks, then on day 2 all tasks dependent on day 0 or day 1 tasks only, and so on… The algorithm maintains 1) a queue of tasks with zero uncompleted dependencies and 2) an array with the number of uncompleted dependencies for each task. The problem setup involves the initialization of the queue and array (each O(N) operations) and the conversion of the dependency list into an adjacency list which contains the dependents for each task (O(M) operations, M=total number of dependencies). Each iteration of the main loop, a task is popped from the front of the queue, and the number of dependencies for each of that task's dependents is reduced by 1. If the new number is zero, that task is added to the queue and assigned a day = the day of the popped task + 1. Like BFS and DFS, each node is added and popped from the queue once (O(N) operations). Each "dependence" is considered once (rather than twice in generic BFS/DFS), so the total running time is O(N+M). Here, a number of operations have been assumed to be O(1): queue.popleft(), queue.append,len(queue), addition, subtraction, accessing elements in Nd and S. Ultimately, the overall asymptotic running time is, is O(N+M) which is the best we can do since we have to look at every task and dependency at least once.

2) Consider the propagation of a signal with initial amplitude, a_0, through a telecommunication network with N junctions (junctions are numbered from *0* to *N-1*). As a signal propagates between two connected junctions, *i* and *j*, it experiences a loss characterized by L_{ij}. If the amplitude at junction *i* is *a*, the amplitude at junction *j* is $L_{ij}a$. Note that $0 \leq L_{ij} < 1$ and $a_0 > 0$. The signal at a junction can be boosted back to a_0 provided that its amplitude when it reaches the junction exceeds a threshold: $a \geq a_{min}$ with $a_{min} > 0$ a specified threshold that is the same for each junction. If the signal amplitude falls below this threshold when it reaches a junction, it is removed from the network and is **not** considered to have successfully reached the junction.

Network data is provided via an n-element adjacency list, *A*. The *ith* element of *A* is a list containing two-element tuples of the form (j, L_{ij}). if A[3] = [(4,0.5),(0,0.25)] this indicates that junction 3 has connections to two other nodes, junctions 4 and 0, and that $L_{3,4} = 0.5$ and $L_{3,0} = 0.25$. The network is undirected, so $L_{ij} = L_{ji}$.

i) Develop and implement an efficient algorithm to determine if a signal with initial amplitude, a_0, can successfully propagate from junction J_1 to junction J_2. Here, $a_0, J_1, J_2, A, a_{min}$ are all provided as input. Complete the function *findPath* in *p1_dev.py* so that it finds one feasible path (if such a path exists) and returns the path in a list containing a sequence of integers corresponding to the sequence of junctions that that the signal passes through. So, if there is a feasible path between $J_1 = 5$ and $J_2 = 12$ via junctions 3 and 7 (in that order), the function should return [5,3,7,12]. Add a discussion of your implementation to the docstring of your function.

Ans: The solution code is below:

```
    def findPath(A,a0,amin,J1,J2):
        """
```

```
Question 1.2 i)
Search for feasible path for successful propagation of signal
from node J1 to J2

Input:
A: Adjacency list for graph. A[i] is a sub-list containing two-element tuples (the
sub-list my also be empty) of the form (j,Lij). The integer, j, indicates that there is a link
between nodes i and j and Lij is the loss parameter for the link.

a0: Initial amplitude of signal at node J1

amin: If a>=amin when the signal reaches a junction, it is boosted to a0.
Otherwise, the signal is discarded and has not successfully
reached the junction.

J1: Signal starts at node J1 with amplitude, a0
J2: Function should determine if the signal can successfully reach node J2 from node J1

Output:
L: A list of integers corresponding to a feasible path from J1 to J2.

Discussion: Add analysis here
"""

#---Problem setup----------------
N = len(A)
L2 = [0 for i in range(N)] #Explored/Unexplored
L3 = [-1 for i in range(N)] #Distances
Lprev = [-1 for i in range(N)] #Previous node in shortest path (used to construct feasible path)
aratio = amin/a0
Q=deque([])
#----------------------------

#---- Run BFS to find if  feasible path exists ----
Q.append(J1)
L2[J1]=1
L3[J1]=0
Lprev[J1]=[]
L=[]
while len(Q)>0:
    x = Q.popleft()
    for v,w in A[x]:
        if aratio<w: #Extra check to see if loss is sufficiently small
            if L2[v]==0:
                Q.append(v)
                L2[v]=1
                L3[v]=1+L3[x]
                Lprev[v]=x
                if v==J2:
                    print("found J2")
                    Q=[]
                    break
#-----------------------------------

#---- Construct feasible path for cases where such a path exists---
if L2[J2]==1:
    x=J2
    Dist = L3[J2]
    L = [0 for i in range(Dist+1)]
    L[-1] = J2
    for i in range(Dist,0,-1):
        x = Lprev[x]
        L[i-1] = x
#-------------------------------------
```

```
    return L
```

The core of the algorithm is BFS, modified to 1) account for the loss parameter, 2) terminate if J2 is found, and 3) reconstruct a feasible path when one exists. A queue (using collections deque) is used to efficiently carry out the search, and when searching through neighbors of the popped node, only links with amin/a0<Lij are considered. Rather than construct paths for all nodes as they are added to the queue, Lprev[i] simply contains the the neighbor that was popped when node i was appended to the queue. The modifications to BFS to check the loss and construct Lprev are both O(1) operations (a division, comparison, and assignment), so the BFS portion of the code is O(M+N) with N nodes and M links. Constructing the path is O(D) where D is the length of the shortest path between J1 and J2. This is O(N) in the worst case so the overall asymptotic running time for the algorithm is O(M+N).

ii) Now, develop an efficient algorithm to determine the minimum a_0 needed for a signal to successfully propagate from junction J_1 to J_2. A, J_1, J_2, a_{min} are all provided as input. Complete the function *a0min* in *p1_dev.py* so that it finds both $a_{0,min}$ and a feasible path from J_1 to J_2 with $a_0 = a_{0,min}$ (if a path exists). The function should return both $a_{0,min}$ and the computed path (see the function documentation). If no feasible path exists for any a_0, return $a_{0,min} = -1$ and an empty list for the path. Add a discussion of your implementation to the docstring of your function.

Ans: The solution code is below:

```
def a0min(A,amin,J1,J2):
    """
    Question 1.2 ii)
    Find minimum initial amplitude needed for signal to be able to
    successfully propagate from node J1 to J2 in network (defined by adjacency list, A)

    Input:
    A: Adjacency list for graph. A[i] is a sub-list containing two-element tuples (the
    sub-list my also be empty) of the form (j,Lij). The integer, j, indicates that there is a link
    between nodes i and j and Lij is the loss parameter for the link.

    amin: Threshold for signal boost
    If a>=amin when the signal reaches a junction, it is boosted to a0.
    Otherwise, the signal is discarded and has not successfully
    reached the junction.

    J1: Signal starts at node J1 with amplitude, a0
    J2: Function should determine min(a0) needed so the signal can successfully
    reach node J2 from node J1

    Output:
    (a0min,L) a two element tuple containing:
    a0min: minimum initial amplitude needed for signal to successfully reach J2 from J1
    L: A list of integers corresponding to a feasible path from J1 to J2 with
    a0=a0min
    If no feasible path exists for any a0, return output as shown below.

    Discussion: Add analysis here
    """

    #First check if there is a path for the equivalent unweighted graph using BFS
    isPath = findPath2(A,J1,J2)
    if not isPath: return -1,[]

    #Initialize dictionaries
    dinit = -1
    Edict = {} #Explored nodes
    Udict = {} #Unexplored nodes
```

```
    for n in range(len(A)):
        Udict[n] = (dinit,0)
    Udict[J1]=(1,J1)

    prev=0
    #Main search
    while len(Udict)>0:
        #Find node with max Lmin in Udict and move to Edict
        dmax = dinit
        for n,w in Udict.items():
            if w[0]>dmax:
                dmax=w[0]
                nmax=n
        if dmax==dinit: break
        Edict[nmax] = Udict.pop(nmax)
        print("moved node", nmax)
        if nmax==J2: break

        #Update provisional Lmin's for unexplored neighbors of nmax
        for n,w in A[nmax]:
            if n in Udict:
                dcomp = min(dmax,w) #Lmin for path via nmax to n
                if dcomp>Udict[n][0]:
                    Udict[n]=(dcomp,nmax) #Use new Lmin if larger than old Lmin

    #Construct path
    L = deque([])
    L.append(J2)
    J = J2
    while J != J1:
        J = Edict[J][1]
        L.appendleft(J)

    a0min_value = amin/Edict[J2][0]
    return (a0min_value,list(L))
```

This is a modified implementation of Dijkstra's algorithm from lecture. The aim is to find the path between J1 and J2 with the largest Lmin, where Lmin is the minimum Lij on a path from J1 to J1. This is accomplished by maintaining a priority queue which stores the largest possible Lmin for paths passing through "explored" nodes. The node with the largest provisional Lmin is removed from the queue each iteration, and the search terminates if/when J2 is removed. The implementation uses dictionaries as in the Dijkstra code that was provided. The main difference is that when a node's provisional Lmin is updated, the most recently removed node is stored along with Lmin and used to reconstruct the path after the search has completed. The cost of the search is $O(M + N^2)$ as discussed in lecture. With a binary heap-based implementation of the priority queue, this becomes $O((M + N)log_2 N)$ – with most large complex networks, $M \ll N^2$ and there are considerable savings with the use of a heap.

Part 2. (10 pts)

In Part 2, you will develop code to simulate the the spread of an infectious agent through an organism. The organism is modeled as an anatomical network with *N* nodes, and each node corresponds to a very large number of cells. Cells can be transported between some nodes, and we assume that there is a tendency for moving cells to flow towards highly-connected nodes. Three variables characterize the state of a node: S_j, I_j, and V_j which correspond to the fraction of cells at node *j* which are: Spreaders, Infected (but not spreaders), and healthy but Vulnerable to the agent. Our (crude) model for this problem is:

$$\frac{dS_i}{dt} = \alpha I_i - (\gamma+\kappa)S_i + \sum_{j=0}^{N-1}(F_{ij}S_j - F_{ji}S_i)$$
$$\frac{dI_i}{dt} = \theta S_i V_i - (\kappa+\alpha)I_i + \sum_{j=0}^{N-1}(F_{ij}I_j - F_{ji}I_i)$$
$$\frac{dV_i}{dt} = \kappa(1-V_i) - \theta S_i V_i + \sum_{j=0}^{N-1}(F_{ij}V_j - F_{ji}V_i)$$
$$\theta = \theta_0 + \theta_1(1 - sin(2\pi t))$$

The summation terms on the RHS control the transport between nodes while the other terms on the RHS dictate the dynamics within a node. The *flux matrix*, F_{ij} is defined as $F_{ij} = \tau \frac{q_i A_{ij}}{\sum_{k=0}^{N-1} q_k A_{kj}}$ where A_{ij} is the adjacency matrix of the network (which is unweighted and undirected), q_i is the degree of node i, τ is a model parameter, and F_{ij}/τ corresponds to the fraction of cells moving from j which are going to adjacent node, i. Note that $\sum_{i=0}^{N-1} F_{ij} = \tau$.

The parameter α controls conversion of infected cells to spreaders, θ controls conversion of vulnerable cells into infected cells, γ controls the recovery rate of spreaders, κ controls the conversion of all non-vulnerable cells to vulnerable. For questions 1 and 2 below, you can simply fix these parameters as: $\alpha = 50$, $\theta_0 = 80$, $\theta_1 = 105$, $\gamma = 71$, $\kappa = 1.0$.

The initial condition should correspond to only one specified node being infected. All other nodes should have *V=1*, *I=0*, *S=0*.

1. First, consider the case where $\tau = 0$. Complete the function, *model1*, so that it computes a numerical solution for this model on a network provided as input in the form of a networkx graph. The timespan of the simulation and other model parameters are also provided as input – see the documentation in the template file. The node which should be initially infected is also specified as input, and the initial condition for this node should be, *(V,I,S) = (0.1,0.05,0.05)*. The function should return an array containing *S* for the initially-infected node at each time step (including the initial condition). When input parameter *display* is True, the function should also create a figure of this *S* vs time.

Ans: With $\tau = 0$, the initially-infected and uninfected nodes are decoupled from each other, and we only need to solve the ODEs for node x. The code is below:

```
def model1(G,x=0,params=(50,80,105,71,1,0),tf=6,Nt=400,display=False):
    """
    Question 2.1
    Simulate model with tau=0

    Input:
    G: Networkx graph
    params: contains model parameters, see code below.
    tf,Nt: Solutions Nt time steps from t=0 to t=tf (see code below)
    display: A plot of S(t) for the infected node is generated when true

    x: node which is initially infected

    Output:
    S: Array containing S(t) for infected node
    """
    a,theta0,theta1,g,k,tau=params
    tarray = np.linspace(0,tf,Nt+1)
    S = np.zeros(Nt+1)

    #Add code here
    y0 = [0.05,0.05,0.1]

    pi2 = 2*np.pi
    gkm = -(g+k)
```

```
    akm = -(a+k)

    def RHS(y,t):
        th = theta0 + theta1*(1-np.sin(pi2*t))
        dy = [0,0,0]
        dy[0] = a*y[1] + gkm*y[0]
        dy[1] = th*y[0]*y[2] + akm*y[1]
        dy[2] = k*(1-y[2])-th*y[0]*y[2]
        return dy

    y = odeint(RHS,y0,tarray)

    if display:
        plt.figure()
        plt.plot(tarray,y)
        plt.xlabel('t')
        plt.ylabel('S(t)')
        plt.title('Variation of S with time for initially infected node')

    return y[:,0]
```

2. Now, complete *modelN* so that it simulates the model with finite positive τ. The input is the same as in *model1* with the exception that when *display=True*, two figures should be generated: 1) a figure of $< S(t) >$ where <f> represents an average over the *N* nodes of the network and a figure of $< (S(t) - < S(t) >)^2 >$.

i) Complete the function, *RHS*, so that it computes and returns the right-hand side of the model equations (see the function docstring). You should assume that RHS will be called a very large number of times within one call to modelN and construct your code accordingly. You should also design your code for very large networks, say, *N=1e4* or larger, though it is recommended that you develop and test your code with much smaller graphs. Construct an estimate of the number of operations required to compute $dS_i/dt, i = 0, \ldots, N-1$ in one call to RHS. Add this estimate to the docstring of *RHS* along with a concise explanation of how it was constructed.

Ans: The key points here are to: 1) place as many calculations outside of RHS as possible, 2) use built-in numpy functions/methods within RHS where possible, 3) recognize that only the 1st term in each sum on the RHS of the equations needs to be computed (e.g. for the 1st equation, the 2nd term evaluates to $-\tau S_i$), and 4) avoid constructing the full N x N adjacency matrix, A. It is ok to work with A in sparse form, since typically, large complex networks are sparse. However, the problem does not specify that the graph is sparse, so it is also ok to iterate through the links of nodes using the adjacency information stored in G (this will be relatively slow unless A is dense and uses a substantial amount of memory). The solution code is below:

```
def modelN(G,x=0,params=(50,80,105,71,1,0.01),tf=6,Nt=400,display=False):
    """
    Question 2.1
    Simulate model with tau=0

    Input:
    G: Networkx graph
    params: contains model parameters, see code below.
    tf,Nt: Solutions Nt time steps from t=0 to t=tf (see code below)
    display: A plot of S(t) for the infected node is generated when true

    x: node which is initially infected

    Output:
    Smean,Svar: Array containing mean and variance of S across network nodes at
                each time step.
    """
    a,theta0,theta1,g,k,tau=params
    tarray = np.linspace(0,tf,Nt+1)
    Smean = np.zeros(Nt+1)
    Svar = np.zeros(Nt+1)
```

```
#Add code here

#Parameters used in rHS
pi2 = 2*np.pi
gktm = -(g+k+tau)
aktm = -(a+k+tau)
ktm = -(k+tau)


#Set up flux matrix (in sparse form)
N = G.number_of_nodes()
Q = [t[1] for t in G.degree()]

Pden = np.zeros(N)
Pden_total = np.zeros(N)
for j in G.nodes():
    for m in G.adj[j].keys():
        Pden[j] += Q[m]
Pden = 1/Pden
Q = tau*np.array(Q)

F = nx.adjacency_matrix(G)*1.
for i in G.nodes():
    F[i,:] = F[i,:]*Q[i]
    F[:,i] = F[:,i]*Pden[i]

def RHS(y,t):
    """Compute RHS of model at time t
    input: y should be a 3N x 1 array containing with
    y[:N],y[N:2*N],y[2*N:3*N] corresponding to
    S on nodes 0 to N-1, I on nodes 0 to N-1, and
    V on nodes 0 to N-1, respectively.
    output: dy: also a 3N x 1 array corresponding to dy/dt

    Discussion: add discussion here
    """
    th = theta0 + theta1*(1-np.sin(pi2*t))
    dy = np.zeros(3*N)
    S,I,V = y[0:N],y[N:2*N],y[2*N:] #this is a little inefficient, but makes the code comprehensi
    thSV = th*S*V

    dy[:N] = a*I + gktm*S + F.dot(S)
    dy[N:2*N] = thSV + aktm*I + F.dot(I)
    dy[2*N:] = k + ktm*V - thSV + F.dot(V)

    return dy

#Initial conditions
y0 = np.zeros(3*N)
y0[2*N:] = 1
y0[x] = 0.05
y0[N+x] = 0.05
y0[2*N+x] = 0.1

#Simulation
y = odeint(RHS,y0,tarray)

#Post-processing
S = y[:,:N]
Smean = S.mean(axis=1)
Svar = S.var(axis=1)

if display:
    plt.figure()
```

```
        plt.plot(tarray,Smean)
        plt.plot(tarray,Svar)
        plt.legend(('S mean','S var'))
        plt.xlabel('t')
        plt.ylabel('$S_{mean},S_{var}$')
        plt.title('Simulation results for network infection model')

    return Smean,Svar
```

The operation count for computing dS/dt in RHS is: 2N multiplications ($a * I, gktm * S$), and a further $2MN$ operations for the matrix-vector product, F.dot(S) (it would be $2N(N-1)$ if we used a "regular" numpy matrix). There are an additional 2N assignments to construct S and I, but this cost will be negligible compared to the arithmetic operations.

ii) Add the needed code below *RHS* to 1) simulate the model for *Nt* time steps from *t=0* to *t=tf* with the initial condition the same as in *model1*, 2) display the mean and variance of *S* when required, and 3) return the mean and variance of S.

Ans: This is implemented using the *mean* and *var* methods as shown in the code above.

3. Consider the infection model with $\kappa = \alpha = \gamma = \theta_1 = 0$. Investigate the similarities and differences between the resulting dynamics and linear diffusion on Barabasi-Albert graphs. You should design your own approach to the problem, and identify 1-3 key points, carefully discuss them, and produce a few figures illustrating numerical results related to the key points. Save and submit the figures with your codes (with names fig1.png, fig2.png, ...) Add code for generating your figures along with your discussion to the function, *diffusion*. It is sufficient to consider B-A graphs with *n=100* and *m=5* (see networkX documentation). Add code in the *name==main* portion of the code to call *diffusion* and generate the figures you are submitting with your assignment (the figures do not have to be identical due to the randomness of the B-A model).

Ans:

There are several possible points that can be considered – a few are described here. The relevant code is included at the bottom of this document. The linear diffusion model is:

$$\frac{dS_i}{dt} = -\mathcal{D}\sum_{i=0}^{N-1} L_{ij} S_j$$

where $\mathcal{D}$ is the diffusivity constant and L_{ij} is the Laplacian matrix for the network. The equation for S (and for I and V when $\theta = 0$) is:

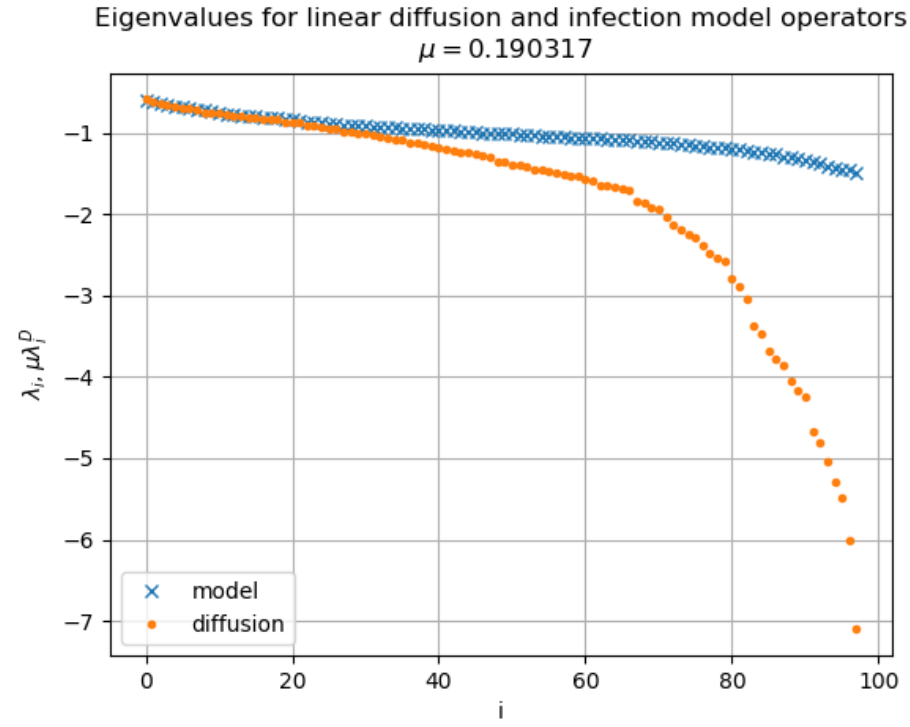
$$\frac{dS_i}{dt} = \tau\sum_{i=0}^{N-1} M_{ij} S_j$$

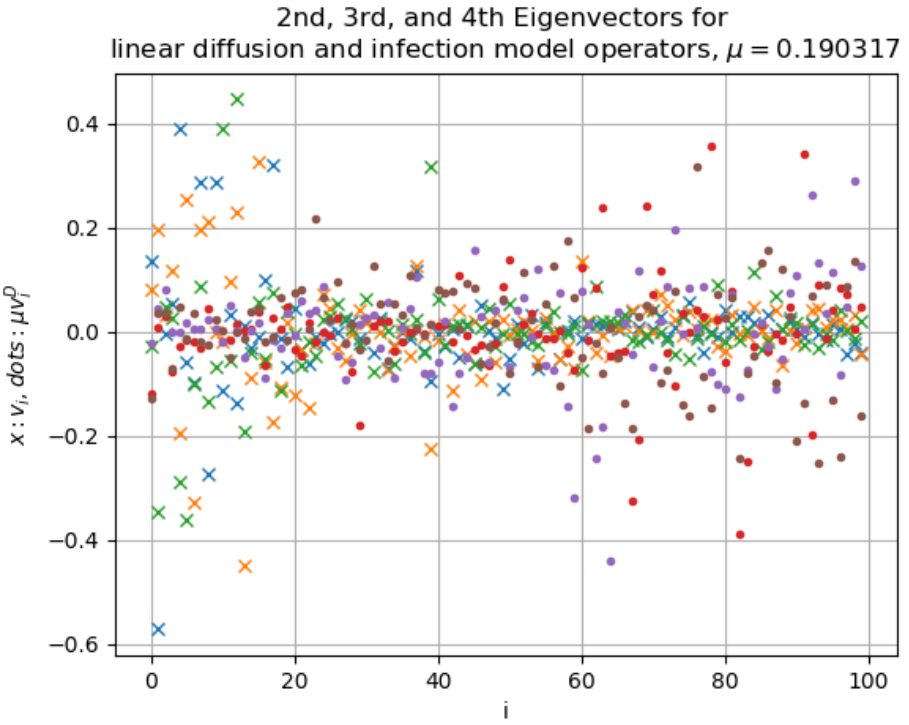
with $M_{ij} = \left(F_{ij}/\tau - \delta_{ij}\right) S_j$. These are linear systems of constant-coefficient ODEs – they are eigenvalue problems, and we first consider the eigenvalues of M and -L which are plotted in the first figure below. Note that the eigenvalues for the diffusion equation have been scaled so that the largest non-zero eigenvalues for the two systems match. A few observations: 1) all shown eigenvalues are real and negative, 2) this indicates that solutions will generally decay in time (monotonically for the diffusion equation since the operator is symmetric), 3) each system also has a zero eigenvalue (not shown) which indicates that the mean of the system stays constant, 4) the eigenvalues for the diffusion eqn. are generally larger in magnitude (faster decay) and span a broader range of values.

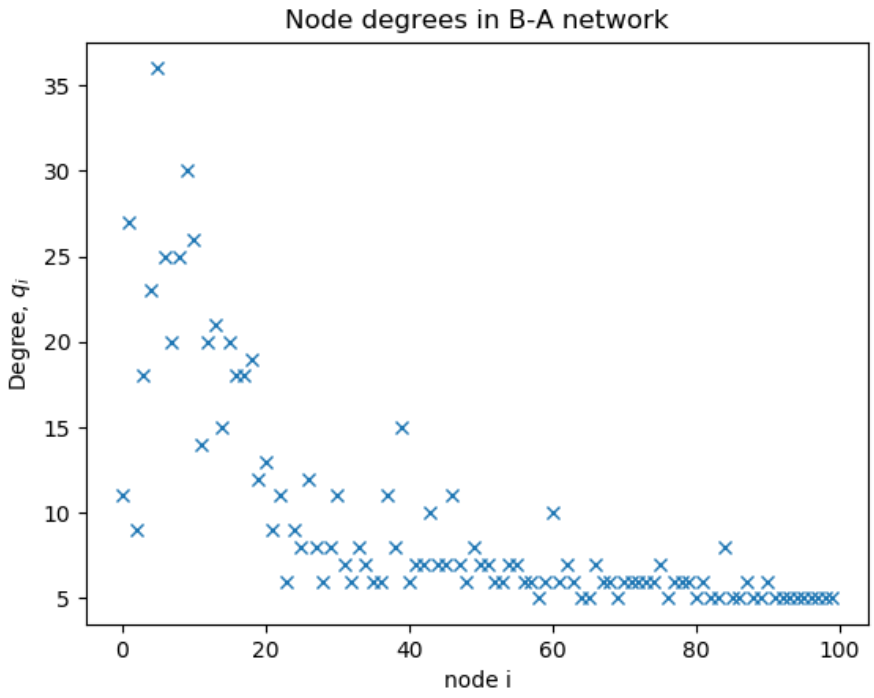
In the comparisons shown below, the diffusivity is set to the factor used to scale the eigenvalues (and τ is set to 1) so that the time-scales of the dynamics are similar. A few eigenvectors (corresponding to the smallest non-zero eigenvalues) are shown in the second figure below. This gives an indication of which nodes are "active", and we see

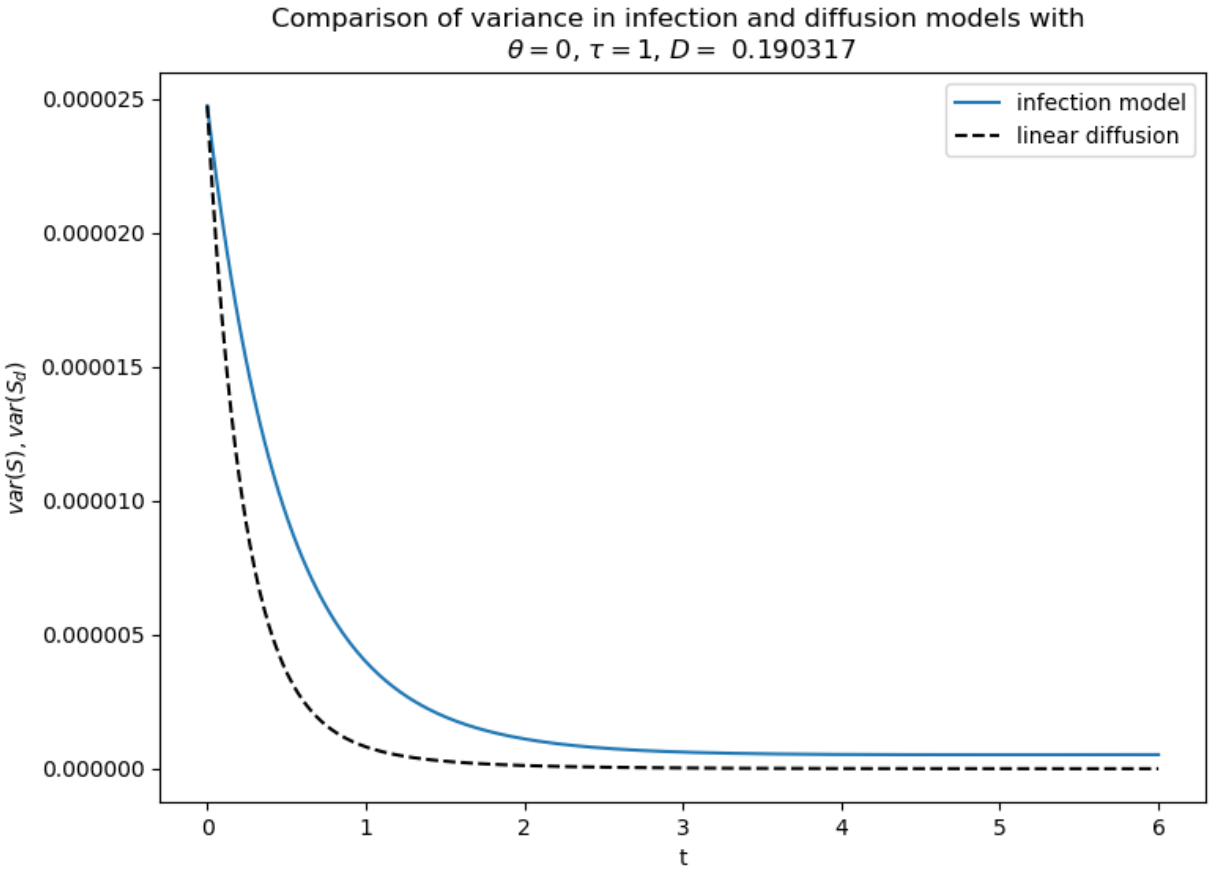
that the diffusion model favors lower-degree nodes, while the infection model favors higher-degree nodes (cf. the 3rd figure).

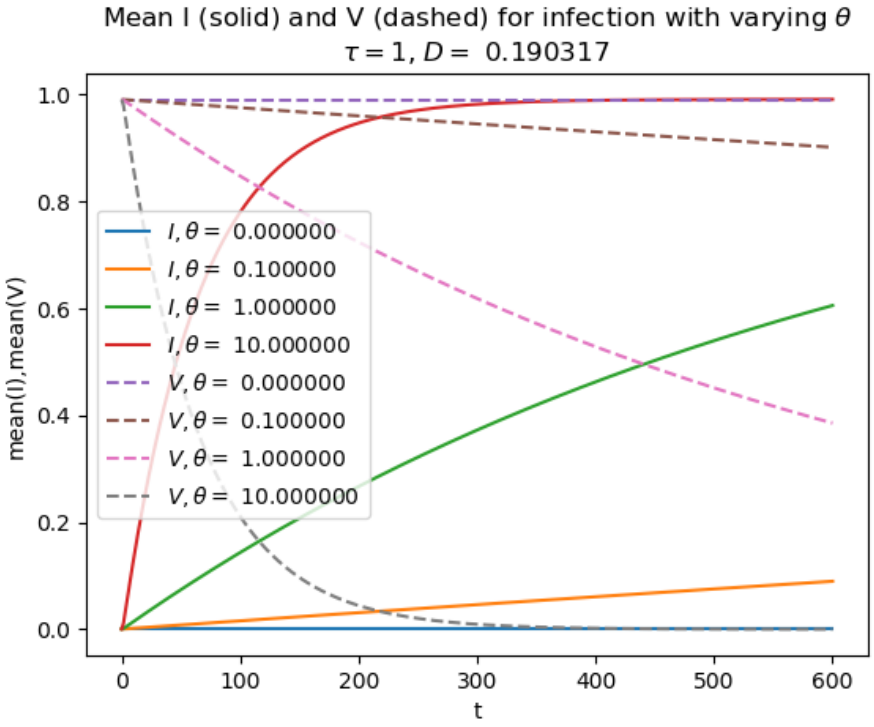
Finally, we consider the evolution of mean and variances of S,I, and V with time. The fourth figure compares the variances of S and $S_{diffusion}$. Note that there is faster decay for the diffusion model which connects clearly to the eigenvalues, while the infection model decays towards a non-zero variance as it favors higher-degree nodes (the eigenvector of the zero-eigenvalue is all ones for the diffusion model but not for the infection model.) The final two figures show the evolution of both the means and variances of I and V for the infection model with varying θ. When $\theta = 0$, the dynamics for S, I, and V are all similar, but finite θ introduces nonlinearity into the system and leads to an increase of I and decrease of V. The rate of increase/decrease increases with θ, and at sufficiently large times both the mean and variance “level off” and approach a state with high I and high variance in I (and low V and low variance in V).

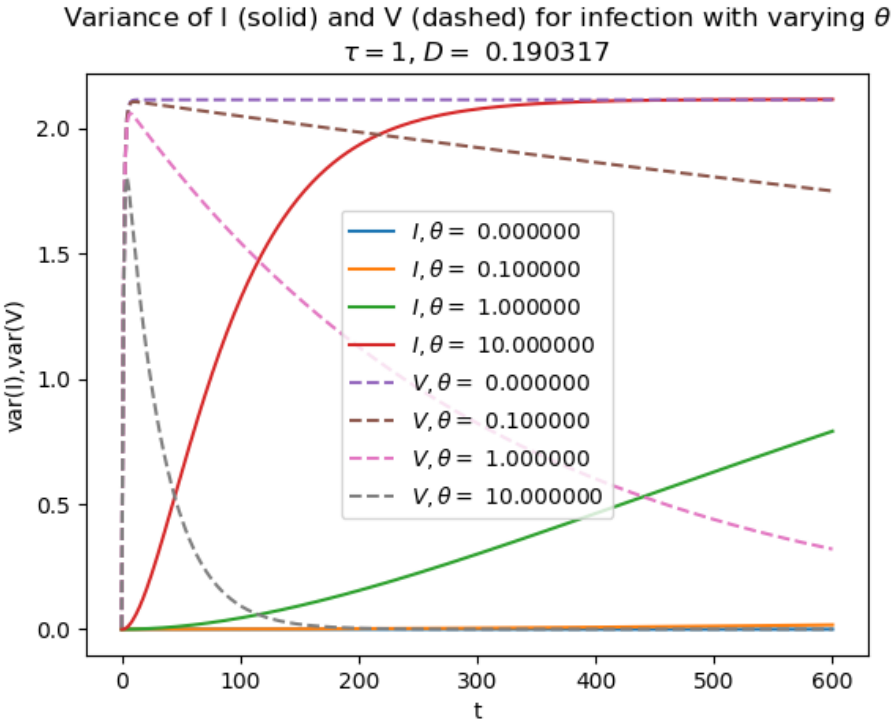

Eigenvalues for linear diffusion and infection model operators
$\mu = 0.190317$
$\lambda_i, \mu\lambda_i^D$
i
model
diffusion
−1
−2
−3
−4
−5
−6
−7
0
20
40
60
80
100


2nd, 3rd, and 4th Eigenvectors for
linear diffusion and infection model operators, $\mu = 0.190317$
$x : v_i, dots : \mu v_i^D$
i
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
20
40
60
80
100


Node degrees in B-A network
Degree, q_i
node i
0
20
40
60
80
100
5
10
15
20
25
30
35


Comparison of variance in infection and diffusion models with
$\theta = 0$, $\tau = 1$, $D =$ 0.190317
infection model
linear diffusion
$var(S), var(S_d)$
t
0.000000
0.000005
0.000010
0.000015
0.000020
0.000025
0
1
2
3
4
5
6






```
def diffusion(tf=6,Nt=400,display=False):
    """
    Question 2.3: Compares simplified infection model with linear diffusion
    for B-A graph with N=100, M=5

    Input:
    tf,Nt: Solutions are generated for Nt time steps from t=0 to t=tf (see code below)
    display: Several plots are generated when true

    Output: none

    """
    tarray = np.linspace(0,tf,Nt+1)
```

```
G = nx.barabasi_albert_graph(100,5,seed=1)
#Set up flux matrix (but without tau)
N = G.number_of_nodes()
Q = [t[1] for t in G.degree()]

Pden = np.zeros(N)
Pden_total = np.zeros(N)
for j in G.nodes():
    for m in G.adj[j].keys():
        Pden[j] += Q[m]
Pden = 1/Pden

A = nx.adjacency_matrix(G)*1.
F = A.copy()
for i in range(N):
    F[i,:] = A[i,:]*Q[i]
for i in range(N):
    F[:,i] = F[:,i]*Pden[i]

M = F - sp.eye(N) #Linear operator for infection model
L = A - sp.diags(Q) # negative Laplacian matrix

#compute eigenvalues and eigenvectors
l,v = sp.linalg.eigs(M,N-2)
ld,vd = sp.linalg.eigsh(L,N-2)

#Sort eigenvalues, eigenvectors
ind = np.argsort(l)
l = l[ind]
v = v[:,ind]

ind = np.argsort(ld)
ld = ld[ind]
vd = vd[:,ind]

fac = np.real(l[-2]/ld[-2])

#set initial conditions, compute integration constants
y0 = np.zeros(3*N)
y0[2*N:] = 1
y0[x] = 0.05
y0[N+x] = 0.05
y0[2*N+x] = 0.1

#c = np.linalg.solve(v,y0[:N])
#c_d = np.linalg.solve(vd,y0[:N])

#Display eigenvalues (scaling diffusion eigenvalues to match 2nd infection eigenvalue)
plt.figure()
plt.plot(l[-1::-1],'x')
plt.plot(fac*ld[-1::-1],'.')
plt.xlabel('i')
plt.ylabel('$\lambda_i, \mu \lambda^D_i$')
plt.grid()
plt.title('Eigenvalues for linear diffusion and infection model operators\n $\mu=$%f' %fac)
plt.legend(('model','diffusion'))

#Display a few eigenvectors
plt.figure()
plt.plot(v[:,-4:-1],'x')
plt.plot(vd[:,-4:-1],'.')
plt.grid()
plt.xlabel('i')
plt.ylabel(r'$x: v_i, dots: \mu v_i^D$')
```

```
plt.title('2nd, 3rd, and 4th Eigenvectors for \n' r'linear diffusion and infection model operator

#Display degree array
plt.figure()
plt.plot(Q,'x')
plt.xlabel('node')
plt.ylabel('Degree')
plt.title('Node degrees in B-A network')


def RHS(y,t,th):
    """Compute RHS of infection model at time t with theta=th
    input: y should be a 3N x 1 array containing with
    y[:N],y[N:2*N],y[2*N:3*N] corresponding to
    S on nodes 0 to N-1, I on nodes 0 to N-1, and
    V on nodes 0 to N-1, respectively.
    output: dy: also a 3N x 1 array corresponding to dy/dt
    """
    dy = np.zeros(3*N)
    S,I,V = y[0:N],y[N:2*N],y[2*N:]
    thSV = th*S*V

    dy[:N] =  F.dot(S) - S
    dy[N:2*N] = thSV + F.dot(I) - I
    dy[2*N:] =  -thSV + F.dot(V) - V

    return dy


def RHS_diffusion(y,t):
    """Compute RHS of linear diffusion model at time t
    input: y should be a 3N element array containing
    S,I,V
    fac (defined above) is used as the diffusivity
    output: dy: also a 3N-element array corresponding to dy/dt
    """
    dy = np.zeros(3*N)
    dy[:N] =  fac*L.dot(y[:N])
    dy[N:2*N] = fac*L.dot(y[N:2*N])
    dy[2*N:] = fac*L.dot(y[2*N:])

    return dy


#Set initial condition
x=0 #Infected node
y0 = np.zeros(3*N)
y0[2*N:] = 1
y0[x] = 0.05
y0[N+x] = 0.05
y0[2*N+x] = 0.1

#Simulations
theta_array = [0,0.1,1,10]
Nth = len(theta_array)
y = np.zeros((Nt+1,3*N,Nth))
y[:,:,0] = odeint(RHS,y0,tarray,args=(0,)) #first with theta=0

#Diffusion solution
yd = odeint(RHS_diffusion,y0,tarray)

#analyze S
S = y[:,:N,0]
Sm = S.mean(axis=1)
Sv = S.var(axis=1)
```

```
Sd = yd[:,:N]
Sdm = Sd.mean(axis=1)
Sdv = Sd.var(axis=1)

plt.figure()
plt.plot(tarray,Sv,label='infection model')
plt.plot(tarray,Sdv,'k--',label='linear diffusion')
plt.xlabel('t')
plt.ylabel('$var(S), var(S_d)$')
plt.title('Comparison of variance in infection and diffusion models with \n' r'$\theta=0$, $\tau=
plt.legend()

#analyze I and V with range of thetas
tarray = tarray*100 #need longer timespan than for S
for i,th in enumerate(theta_array):
    y[:,:,i] = odeint(RHS,y0,tarray,args=(th,))
I = y[:,N:2*N]
Im = I.mean(axis=1)
Iv = I.var(axis=1)

Id = yd[:,N:2*N]
Idv = Id.var(axis=1)

V = y[:,2*N:]
Vm = V.mean(axis=1)
Vv = V.var(axis=1)

Vd = yd[:,2*N:]
Vdv = Vd.var(axis=1)

plt.figure()
for i,th in enumerate(theta_array):
    plt.plot(tarray,Im[:,i],label=r'$I, \theta=$ %f'%th)
for i,th in enumerate(theta_array):
    plt.plot(tarray,Vm[:,i],'--',label=r'$V, \theta=$ %f'%th)

plt.legend()
plt.xlabel('t')
plt.ylabel('mean(I),mean(V)')
plt.title('Mean I (solid) and V (dashed) for infection with varying ' r'$\theta$' '\n' r'$\tau=1$

plt.figure()
for i,th in enumerate(theta_array):
    plt.plot(tarray,Iv[:,i],label=r'$I, \theta=$ %f'%th)
for i,th in enumerate(theta_array):
    plt.plot(tarray,Vv[:,i],'--',label=r'$V, \theta=$ %f'%th)


plt.legend()
plt.xlabel('t')
plt.ylabel('var(I),var(V)')
plt.title('Variance of I (solid) and V (dashed) for infection with varying ' r'$\theta$' '\n' r'$

return None
```