In [ ]:
```
from scipy.io import loadmat
from scipy.linalg import eigh, inv
import numpy as np
import matplotlib.pyplot as plt
```

Load graph dataset

In [ ]:
```
# load gene network
gene_network = loadmat('data/genetics/geneNetwork_rawPCNCI.mat')
A = gene_network['geneNetwork_rawPCNCI'].astype(np.int32)

# load signal dataset
signals = loadmat('data/genetics/signal_mutation.mat')
X = signals['signal_mutation'].T.astype(np.float32)

# load phenotypes (labels)
phenotypes = loadmat('data/genetics/histology_subtype.mat')
y = phenotypes['histology_subtype']

'Shapes: A: {}, X: {}, y: {}'.format(A.shape, X.shape, y.shape)
```

Out[ ]:
```
'Shapes: A: (2458, 2458), X: (2458, 240), y: (240, 1)'
```

a) Distinguishing power

In [ ]:
```
# Compute Laplacian as shift matrix
D = np.diag(A.sum(axis=1))
L = D - A
S = L

# Diagonalize S (reorder evals from largest to smallest)
w, V = eigh(S)
W = np.diag(w)

# verify diagonalization (should be near 0, due to float errors)
print('L1 norm between L and V @ W @ V.T: {}'.format((L - np.dot(np.dot(V, W), V.T)).su
```

```
L1 norm between L and V @ W @ V.T: 1.305906509562601e-11
```

In [ ]:
```
X_gft = V.T @ X # x_gft[i, j] is coefficient of ith freq for jth sample

# verify GFT is valid (should be near 0)
print('Error after V @ V.T @ X: {}'.format(np.linalg.norm(X - (V @ X_gft))))
```

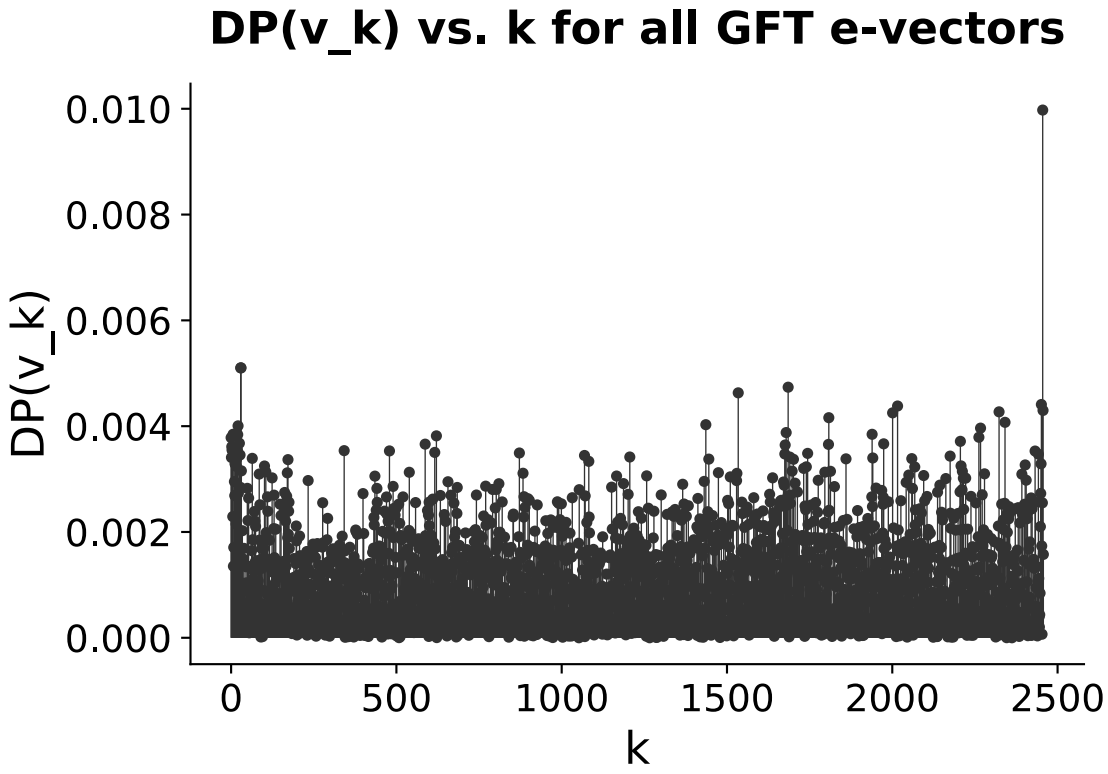
```
Error after V @ V.T @ X: 2.510261873768074e-13
```

In [ ]:
```
# label masks
mask_1 = (y == 1).astype(int).reshape(y.shape[0])
mask_2 = (y == 2).astype(int).reshape(y.shape[0])

# mean filter for each label
mean_1 = mask_1 / mask_1.sum()
mean_2 = mask_2 / mask_2.sum()
```

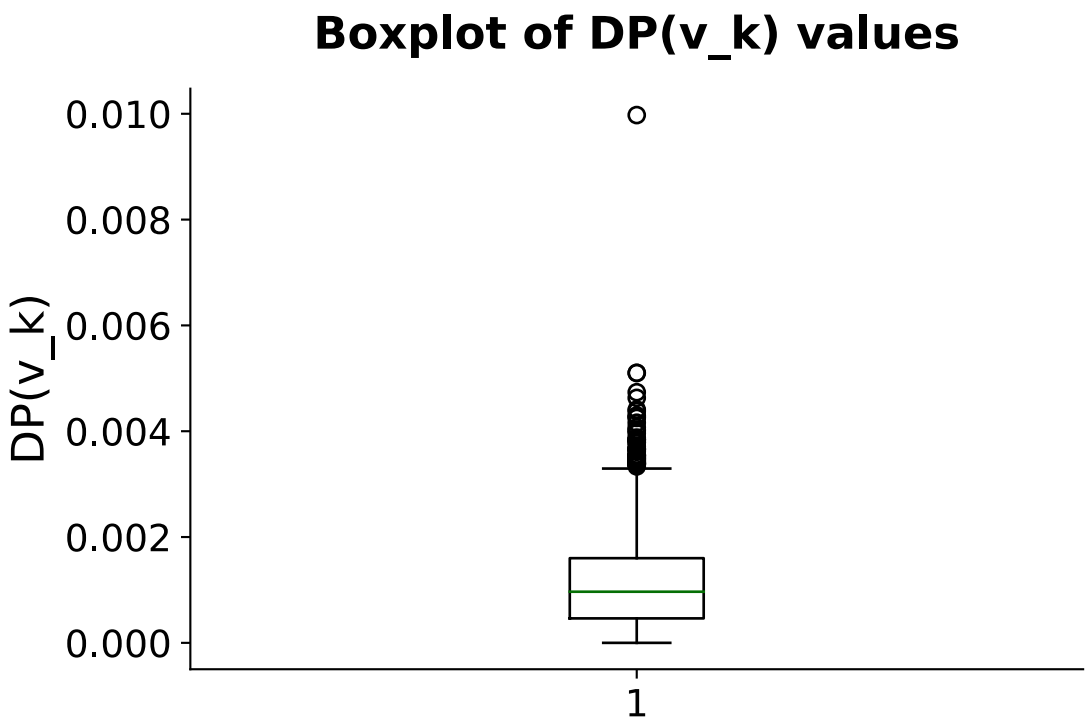
```
# L1 norm of each frequency
k_L1 = np.linalg.norm(X_gft, ord=1, axis=1)
```

In [ ]:
```
DP = np.absolute((X_gft @ mean_1) - (X_gft @ mean_2)) / k_L1
DP = DP.reshape(DP.shape[0])
```

In [ ]:
```
plt.scatter(range(DP.shape[0]), DP, s=10)
plt.vlines(range(DP.shape[0]), 0, DP, linestyle="solid", linewidths=0.5)
plt.xlabel('k')
plt.ylabel('DP(v_k)')
plt.title('DP(v_k) vs. k for all GFT e-vectors');
```



In [ ]:
```
plt.boxplot(DP)
plt.title('Boxplot of DP(v_k) values')
plt.ylabel('DP(v_k)');
```



b) kNN classifier

In [ ]:

```
def knn(X, y, k_all):
    """
    Computes kNN accuracy using leave-one-out cross-validation,
        for several values of k
    """
    # compute pairwise Euclidean distances between all samples
    dists_all = np.full((X.shape[1], X.shape[1]), np.inf)
    for i in range(X.shape[1]):
        for j in range(i+1, X.shape[1]):
            if i == j:
                continue
            dist = np.sqrt(((X[:, i] - X[:, j]) ** 2).sum())
            dists_all[i, j] = dist
            dists_all[j, i] = dist

    # get nearest neighbors of each sample
    nns = np.argsort(dists_all, axis=1)

    # compute cross-val accuracy
    for k in k_all:
        correct = 0
        for col in range(len(X[0])):
            knns = nns[col, :k]
            pred = ((y[knns, 0]).mean() > 1.5) + 1

            correct += pred == y[col, 0]

        print('\tAccuracy using k = {0}: {1:.4f}'.format(k, correct / X.shape[1]))
```

In [ ]:

```
print('Using all frequencies:')
knn(X, y, [3, 5, 7])
```

```
Using all frequencies:
        Accuracy using k = 3: 0.8833
        Accuracy using k = 5: 0.8833
        Accuracy using k = 7: 0.8542
```

c) Filtering almost all frequencies

In [ ]:

```
def filter_gft(X_gft, k_idxs, keep_ratio):
    """
    Filters the GFT of a signal matrix to preserve only the top keep_ratio frequencies,
        as ordered by k_idxs

    X_gft: the GFT of signal matrix X
    k_idxs: priority order with which to keep frequencies
    keep_ratio: ratio of frequency coefficients to preserve in the signal GFT

    Returns X_gft, but with coefficients for frequencies outside the top keep_ratio set
    """
    # select frequencies to keep
    n_keep = int(len(k_idxs) * keep_ratio)
    k_top = k_idxs[:n_keep]
    print('\tKept {} frequencies'.format(n_keep))

    # set all other frequency coefficients to 0
    X_gft_f = np.zeros(X_gft.shape)
    X_gft_f[k_top, :] = X_gft[k_top, :]

    return X_gft_f
```

In [ ]:

```
def filtered_knn(X_gft, y, k_idxs, keep_ratio, n_nbrs_all):
    """
    Filters the GFT of a signal matrix, then evaluates classification accuracy using kN

    X_gft: the GFT of signal matrix X
    y: labels vector
    k_idxs: priority order with which to keep frequencies
    keep_ratio: ratio of frequency coefficients to preserve in the signal GFT
    n_nbrs_all: the different values of k to try for kNN
    """

    # preserve only values at the top frequency
    X_gft_f = filter_gft(X_gft, k_idxs, keep_ratio)

    # take iGFT of filtered signal
    X_f = V @ X_gft_f

    # kNN
    knn(X_f, y, n_nbrs_all)
```

In [ ]:

```
# order frequencies by DP
k_idxs = np.argsort(DP)[::-1]
print('k = {} maximizes distinguishing power'.format(k_idxs[0]))

# try kNN, keeping only the top frequency
filtered_knn(X_gft, y, k_idxs, 1 / len(k_idxs), [3, 5, 7])
```

```
k = 2455 maximizes distinguishing power
        Kept 1 frequencies
        Accuracy using k = 3: 0.9000
        Accuracy using k = 5: 0.8875
        Accuracy using k = 7: 0.8917
```

Accuracy is higher using only one frequency than with all frequencies! This means the original signals are very noisy (in terms of the labels provided).

In [ ]:

```
# try kNN, with other values of p
p_all = [0.75, 0.8, 0.85, 0.9, 0.95]

for p in p_all[::-1]:
    print('With top {} of frequencies (p = {}):'.format(np.round(1 - p, decimals=2), p))
    filtered_knn(X_gft, y, k_idxs, 1 - p, [3, 5, 7])
```

```
With top 0.05 of frequencies (p = 0.95):
        Kept 122 frequencies
        Accuracy using k = 3: 0.9167
        Accuracy using k = 5: 0.9167
        Accuracy using k = 7: 0.9167
With top 0.1 of frequencies (p = 0.9):
        Kept 245 frequencies
        Accuracy using k = 3: 0.9167
        Accuracy using k = 5: 0.9250
        Accuracy using k = 7: 0.9250
With top 0.15 of frequencies (p = 0.85):
        Kept 368 frequencies
        Accuracy using k = 3: 0.9125
        Accuracy using k = 5: 0.9125
        Accuracy using k = 7: 0.9208
With top 0.2 of frequencies (p = 0.8):
        Kept 491 frequencies
        Accuracy using k = 3: 0.9208
        Accuracy using k = 5: 0.9167
        Accuracy using k = 7: 0.9167
With top 0.25 of frequencies (p = 0.75):
        Kept 614 frequencies
        Accuracy using k = 3: 0.9125
        Accuracy using k = 5: 0.9167
        Accuracy using k = 7: 0.9167
```

Accuracy is slightly higher with more frequencies vs. one frequency, and noticeably higher vs. all frequencies, peaking at around the top ~10% of frequencies. This means that, the frequencies below the 90th percentile (in terms of DP) generally contribute mostly noise, relative to the task of predicting patient subtypes.

In [ ]:
```
import networkx as nx
from scipy.io import loadmat
from scipy.linalg import eigh
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

a) Construct a network from the flight data

In [ ]:
```
# load data
node_metadata = pd.read_csv('data/epidemics/airport_Nodes_GC.csv')
edge_data = pd.read_csv('data/epidemics/airport_Edges_GC.csv')

# map given node IDs to index order
node_id_map = {given_id: node_id for (given_id, node_id) in zip(node_metadata.Id, node_
```

In [ ]:
```
# build directed adjacency matrix from edge data
A_dir = np.zeros((len(node_id_map), len(node_id_map)))

for i, row in edge_data.iterrows():
    u, v = node_id_map[row.Source], node_id_map[row.Target]
    A_dir[u][v] = row.Weight

# Compute undirected A
A = (A_dir + A_dir.T) / 2
G = nx.from_numpy_matrix(A)
```

In [ ]:
```
# confirm graph is connected
print('Graph is connected: {}'.format(nx.is_connected(G)))
```

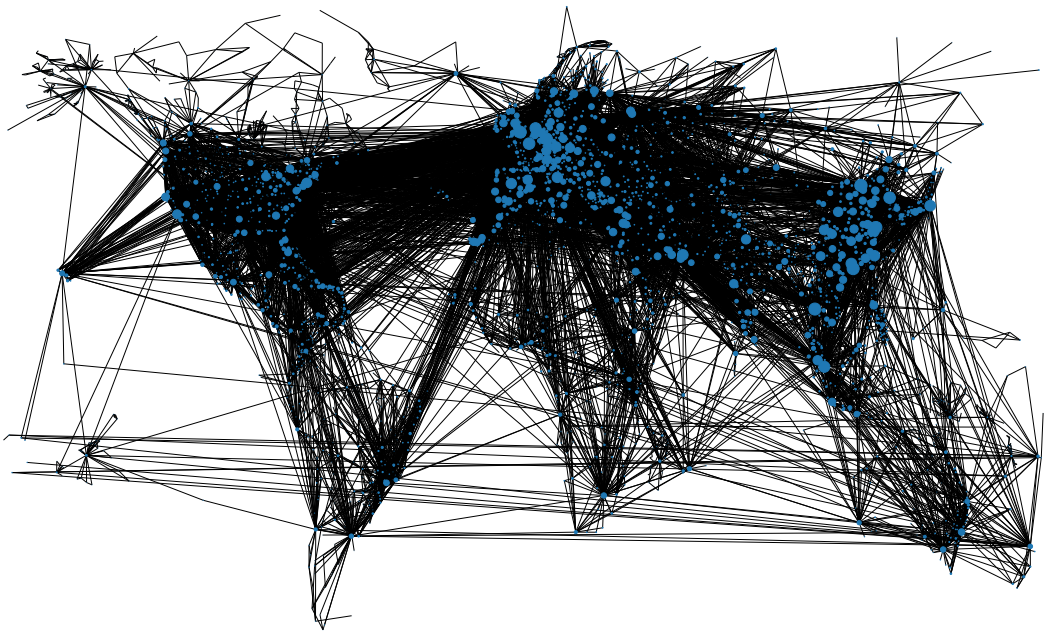
```
Graph is connected: True
```

b) Plot the airport network

In [ ]:
```
# Compute top 2 e-val centralities as sanity check
evec_centralities = nx.centrality.eigenvector_centrality(G, weight='weight')
'Top 2 e-vec centralities: {}'.format(list(sorted(evec_centralities.values()))[-2:])
```

Out[ ]:
```
'Top 2 e-vec centralities: [0.16991022532548966, 0.1785292296913075]'
```

In [ ]:
```
# plot the network, using lat/long as node locations and e-vector centrality as node si
pos = {idx: (row.Longitude, row.Latitude) for idx, row in node_metadata.iterrows()}
node_size = [evec_centralities[n] * 1750 for n in G.nodes()]
plt.figure(1, figsize=(25, 15))
nx.draw(G, pos=pos, node_size=node_size, width=0.25)
```



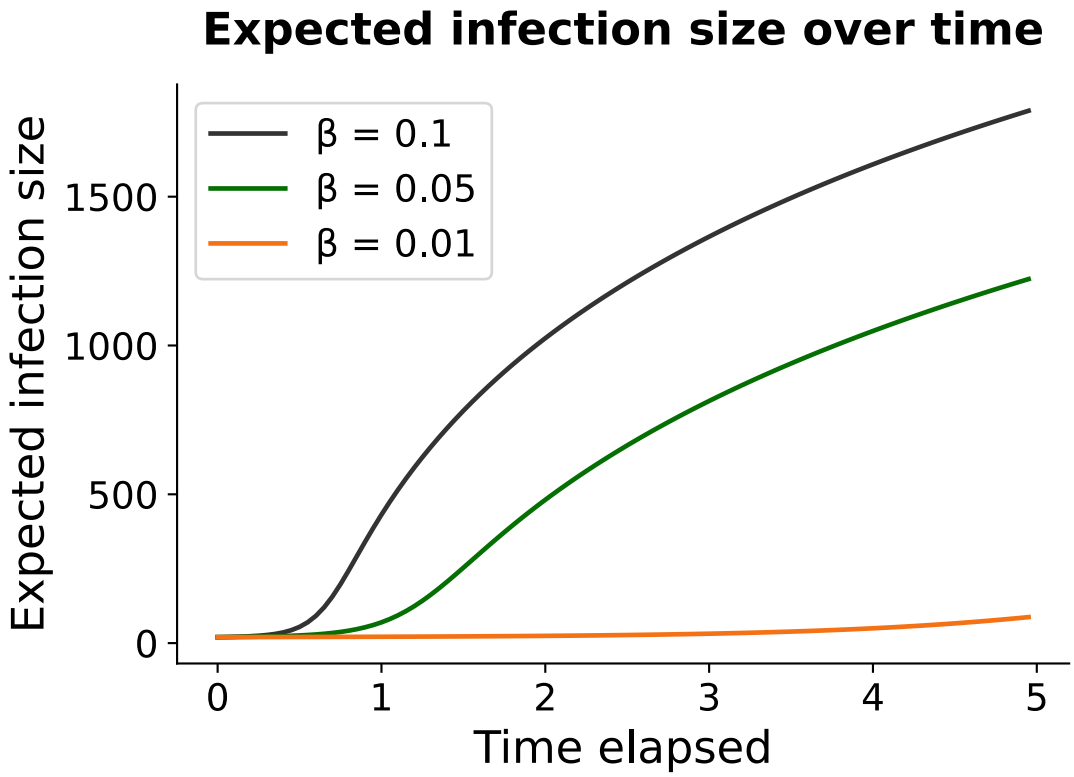
d) Mean-field approximation

In [ ]:

```
def dp_dt(A, p_t, beta, gamma):
    """
    The provided formula for dp/dt, computed using matrix operations
    """
    left = beta * (1 - p_t)
    right = (A @ p_t) - (gamma * p_t)
    return left * right

def infect(A, beta, gamma, dp_dt):
    """
    Models the spread of infection throughout a graph over time

    beta, gamma: model parameters
    dp_dt: function to compute change in infection probabilities across nodes

    Returns P; P[a][b] = infection probability for ath node at bth timestep
    """
    # infect the first 20 nodes
    p_0 = np.zeros((len(A),))
    p_0[:20] = 1

    # compute infection probabilities over time
    t_max = 5
    delta = 0.05

    P = np.zeros((len(p_0), int(t_max / delta)))
    P[:, 0] = p_0

    for i in range(1, int(t_max / delta)):
        P[:, i] = P[:, i-1] + (delta * dp_dt(A, P[:, i-1], beta, gamma))

    return P
```

In [ ]:
```
gamma = 0.1

# test varying values of beta
P_bar_all = []
for beta in (0.1, 0.05, 0.01):
    P = infect(A, beta, 0.1, dp_dt)
    P_bar_all.append(P.sum(axis=0))

# plot expected infected size over time for each beta value
x = np.array(range(0, int(5 / 0.05))) * 0.05
[plt.plot(x, P_bar) for P_bar in P_bar_all]
plt.legend(["β = 0.1", "β = 0.05", "β = 0.01"])
plt.title("Expected infection size over time")
plt.xlabel("Time elapsed")
plt.ylabel("Expected infection size")
```

Out[ ]:
```
Text(0, 0.5, 'Expected infection size')
```



e) Minimum number of immunizations

In [ ]:
```
def immunize(A, node_order, eval_thresh, start_at=0):
    """
    Iteratively remove nodes, in the given order, until the largest e-value of the grap
        drops below the given threshold

    A: adjacency matrix
    node_order: order in which to remove nodes
    eval_thresh: terminate when largest e-val drops below this threshold
    start_at: remove this many nodes before starting to compute largest e-val (saves ti
    """
```

```
    A_imm = A.copy()

    for i in range(len(node_order)):
        # remove a node
        n = node_order[i]

        # setting to 0 creates additional e-vals that are 0, but still produces valid l
        #   this is easier than deleting rows/columns
        A_imm[n, :] = 0
        A_imm[:, n] = 0

        if i < start_at:
            continue
        elif i == start_at:
            print('Starting with {} nodes removed'.format(i))

        # recompute max e-val
        w_max = eigh(A_imm, eigvals_only=True, eigvals=(len(A_imm) - 1, len(A_imm) - 1)

        if w_max < eval_thresh:
            print('Max e-val after removing {} nodes: {}'.format(i+1, w_max))
            print('Max e-val below threshold after removing {} nodes'.format(i+1))
            break

        if i % 5 == 4:
            print('Max e-val after removing {} nodes: {}'.format(i+1, w_max))

    return A_imm
```

In [ ]:

```
# compute required threshold for λ_max(A)
beta = 0.01
gamma = 0.4

print('Must have λ_max(A) < {}'.format(gamma / beta))
```

```
Must have λ_max(A) < 40.0
```

In [ ]:

```
# compute degree centrality
degree_centralities = A.sum(axis=1) / (len(A) - 1)

# order nodes by degree centrality
node_order = np.argsort(degree_centralities)[::-1]

# compute number of nodes necessary to remove
immunize(A, node_order, gamma / beta, start_at=50); # start with 50 to speed up re-comp
```

```
Starting with 50 nodes removed
Max e-val after removing 55 nodes: 45.751362453969314
Max e-val after removing 60 nodes: 45.749493453262154
Max e-val after removing 65 nodes: 45.74924054241361
Max e-val after removing 70 nodes: 41.151565542207784
Max e-val after removing 73 nodes: 37.41654106581537
Max e-val below threshold after removing 73 nodes
```

In [ ]:

```
# compute e-val centrality
eval_centralities_dict = nx.centrality.eigenvector_centrality(G, weight='weight')
eval_centralities = np.array([eval_centralities_dict[n] for n in G.nodes()])
```

```
# order nodes by e-val centrality
node_order = np.argsort(eval_centralities)[::-1]

# compute number of nodes necessary to remove
immunize(A, node_order, gamma / beta, start_at=100);
```

```
Starting with 100 nodes removed
Max e-val after removing 105 nodes: 48.51744278279449
Max e-val after removing 110 nodes: 48.51591111485003
Max e-val after removing 115 nodes: 48.51590896694197
Max e-val after removing 120 nodes: 48.51590773253301
Max e-val after removing 125 nodes: 39.427119599712015
Max e-val below threshold after removing 125 nodes
```

Degree centrality only required 73 nodes to be removed, while e-val centrality required 125 nodes to be removed.

ELEC 573 HW 2

Nicholas Glaze
n1g2

2 c) Let $P(X_i(t+\Delta)=1 \mid X_i(t)=0, x(t)) = \rho_i$

show that $\sum_{j \in N_i} \beta a_{ij} X_j(t)$ is an upper bound for $1-\prod_{j \in N_i \wedge X_j(t)=1}(1-\beta a_{ij}) = \rho$

- Let node i have n nbrs j s.t. $X_j(t)=1$;

a) then since $(1-\beta a_{ij}) = 1$ when $a_{ij}=0$, $\rho = 1-(1-\beta)^n$
$+(1-\beta)$ when $a_{ij}=1$

b) and $\sum_{j \in N_i} \beta a_{ij} X_j(t) = \underline{\beta \cdot n}$, since $a_{ij} X_j(t) = 1$ for n nodes, else 0

$\Rightarrow \beta n \overset{?}{\geq} 1-(1-\beta)^n$ (must have $n \in \mathbb{N}$, $0 \leq \beta \leq 1$)

$\star \; 1 \overset{?}{\leq} \beta n + (1-\beta)^n$

$\frac{\partial}{\partial \beta}(\cdot) = n - n(1-\beta)^{n-1} = 0 \rightarrow$ $n=0 \rightarrow \beta + \ln(1-\beta) = 0$

$\ln(1-\beta) = -\beta$

$e^{-\beta} = 1-\beta$

$\beta = 0$

$1-(1-\beta)^{n-1} = 0$

$1 = (1-\beta)^{n-1}$

$\beta = 0$ or $n = 1$

$0 + 1^n \ln(1) = 0$

$0 = 0 \Rightarrow$ any n

$\frac{\partial}{\partial n}(\cdot) = \beta + (1-\beta)^n \ln(1-\beta) = 0$

$\beta + (1-\beta)\ln(1-\beta) = 0$

$\ln(1-\beta) = \frac{-\beta}{1-\beta}$

$e^{-\beta/1-\beta} = 1-\beta$

$e^{\beta/\beta-1} = 1-\beta \Rightarrow$ only $\beta = 0$

Local minima:

$(n=0, \beta=0) \rightarrow 1 \leq 0+1$ ✓

$(\beta=0, \text{any } n) \rightarrow 1 \leq 0+1$ ✓

$(n=1, \beta=0) \rightarrow 1 \leq 0+1$ ✓

$\beta n + (1-\beta)^n$ has minimum value 1, $(n \in \mathbb{N}, 0 \leq \beta \leq 1)$

so $1 \leq \beta n + (1-\beta)^n$ is true $\Rightarrow \sum_{j \in N_i} \beta a_{ij} X_j(t)$

is a valid upper bound for ρ. □