Week 3 : SIR Process

Imagine a disease. Nodes are in 3 states, S/I/R When a node is I, they will remain I for 1 week and then be R. When a node is I, they will cause their adjacent nodes to be I with probability λ, independently. A person who is R will no longer partake in any dynamic process.

To start the outbreak, define an initial condition. Almost all the nodes are S, a small fraction of nodes chosen uniformly at random, begin as I.

Terminal state is when we only have nodes of S/R and no I left.

Assumptions:

We assume no one is added to S group, the only way a node leaves the S group is by becoming I. Assumed a fixed fraction k of the infected group will recover during any given day. In this case since duration of infection is 3 days then $\frac{1}{7}$.

[Differential Relationships] : https://maa.org/press/periodicals/loci/joma/the-sir-model-for-spread-of-disease-the-differential-equation-model

[Implementing in Python] : https://pythonhosted.org/epydemic/sir.html#:~:text=The%20Susceptible%2DInfected%2DRemo

[Network Analysis to Identify the Risk of Epidemic Spreading] : https://www.mdpi.com/2076-3417/11/7/2997#:~:text=where%20S%2C%20I%2C%20and%20R,the%20whole%20population%2

[Two critical times for the SIR model] : https://bpb-us-w2.wpmucdn.com/web.sas.upenn.edu/dist/6/47/files/2021/07/1-s2.0-S0022247X21005862-main.pdf

[Exploring the threshold of epidemic spreading for a stochastic SIR model with local and global contacts] : https://www.sciencedirect.com/science/article/pii/S0378437119318035#:~:text=The%20threshol

Q1

Generating model with nodes n and mean k :

Motivation for the Configuration Model over the Random Model

The configuration model is a model in which the degrees of vertices are fixed beforehand. Such a model is more flexible than the generalized random graph. For example, the generalized random graph always has a positive proportion of vertices of degree 0, 1, 2, etc. . In some real-world networks, however, it is natural to investigate graphs where every vertex has at least one or two neighbors.

For various λ values, run until only S/R state are left.

In [ ]:
```
import numpy as np
import scipy.stats as scistats
import matplotlib.pyplot as plt
import random
from collections import Counter
from scipy.cluster.hierarchy import DisjointSet
```

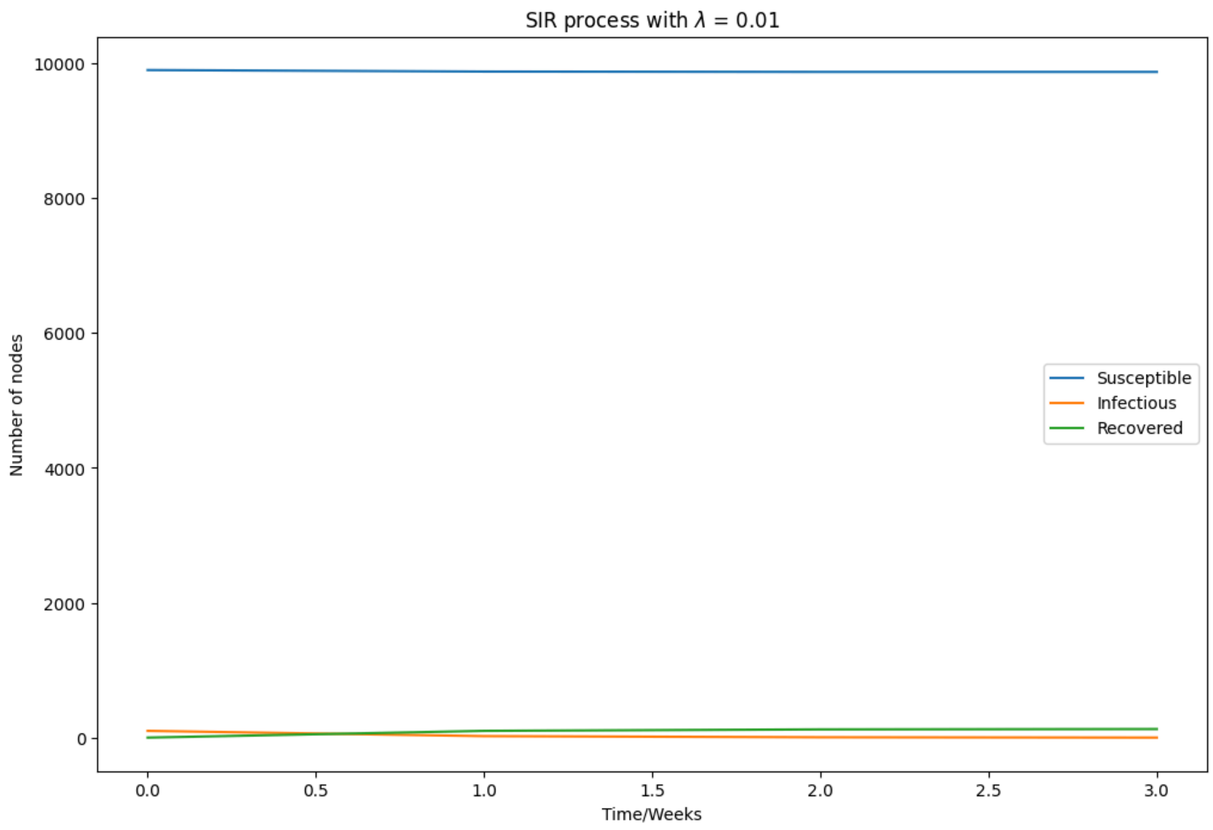
In [ ]:
```
class Network():
    def __init__(self,num_nodes):
        self.adj = {i:set() for i in range(num_nodes)}
        self.num_edge = 0
        self.num_nodes = num_nodes

    def add_edge(self,i,j):
        self.adj[i].add(j)
        self.adj[j].add(i)
        self.num_edge+=1

    def neighbors (self,i):
        return self.adj[i]

    def edge_list(self):
        return [(i,j) for i in self.adj for j in self.adj[i] if i<j]
```

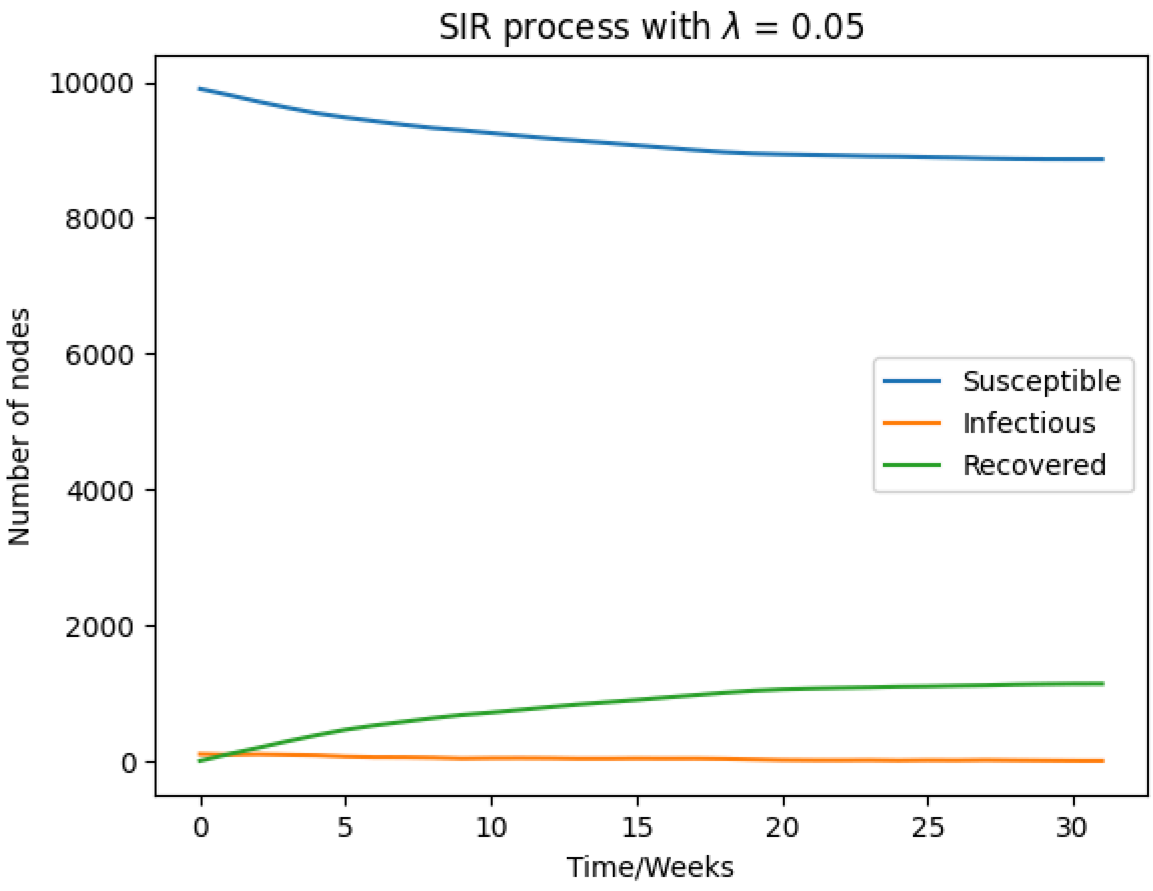
In [ ]:
```
class Erdos_renyi_Network(Network):

    def __init__(self, num_nodes, mean):
        super().__init__(num_nodes)

        # Parameter p for a Erdos-renyi Graph
        self.p = mean/(num_nodes-1)

        # Construct Erdos-renyi Graph
        for i in range(num_nodes):
            for j in range(i+1,num_nodes):
                if np.random.random()<self.p:
                    self.add_edge(i,j)
```

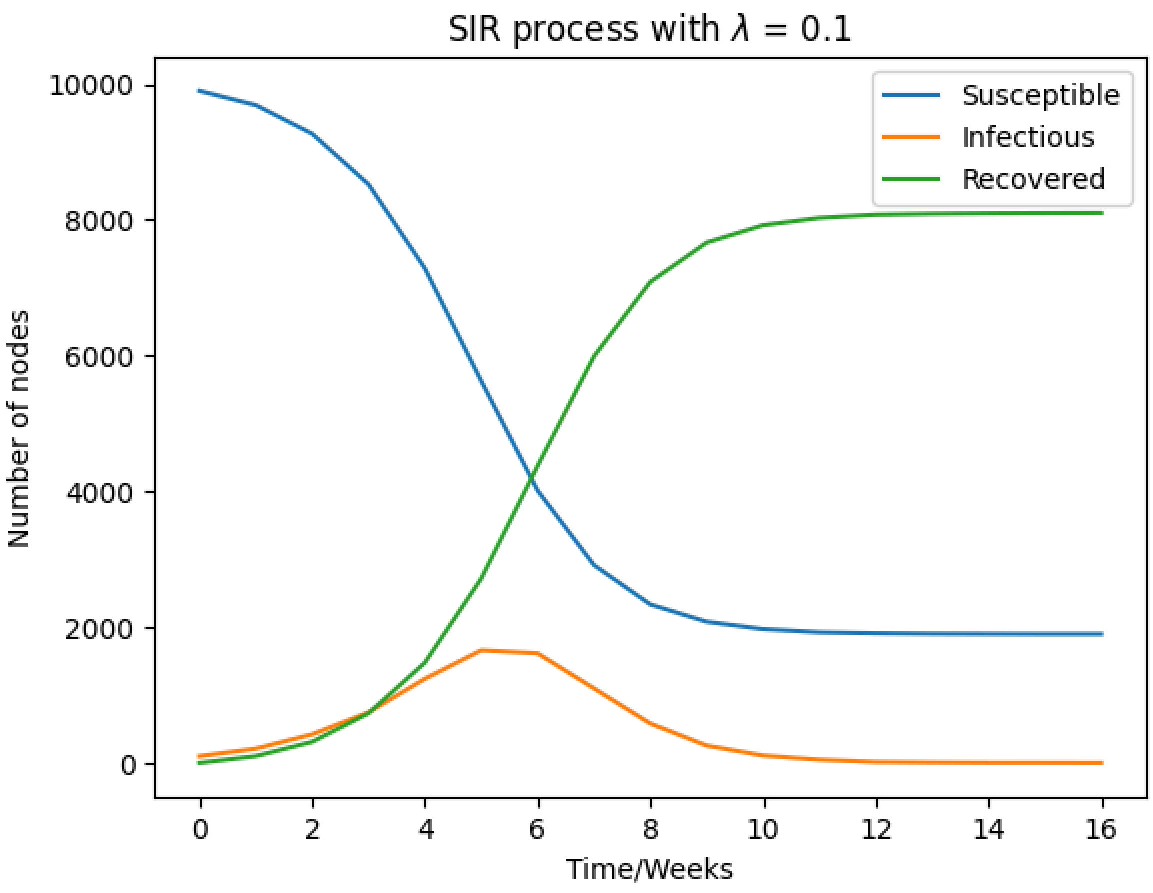
In [ ]:
```
class SIR_Model():

    def __init__(self, network: Erdos_renyi_Network, p_infected,p_infect):

        self.p_infected = p_infected
        self.p_infect = p_infect
        self.network = network

        # SIR nodes
        self.S = {node for node in range(self.network.num_nodes)}
        self.I = set()
        self.R = set()

        # Initially infect a small fraction of the population
        self.I.update(np.random.choice(list(self.S), size=int(self.p_infected*self.
        self.S.difference_update(self.I)

    def run(self):
        '''Runs simulation for a cycle'''

        new_I = set()
        for node in self.I:
            for adj in self.network.neighbors(node):
                if adj in self.S and np.random.random()<self.p_infect:
                    new_I.add(adj)

            self.R.add(node)

        self.I.difference_update(self.R)
        self.I.update(new_I)
        self.S.difference_update(self.I)


    def run_to_extinction(self):
        '''Runs simulation until extinction, then returns time series of SIR number

        S_list, I_list, R_list = [len(self.S)], [len(self.I)], [len(self.R)]

        while self.I:

            self.run()

            S_list.append(len(self.S))
            I_list.append(len(self.I))
            R_list.append(len(self.R))

        return S_list, I_list, R_list
```

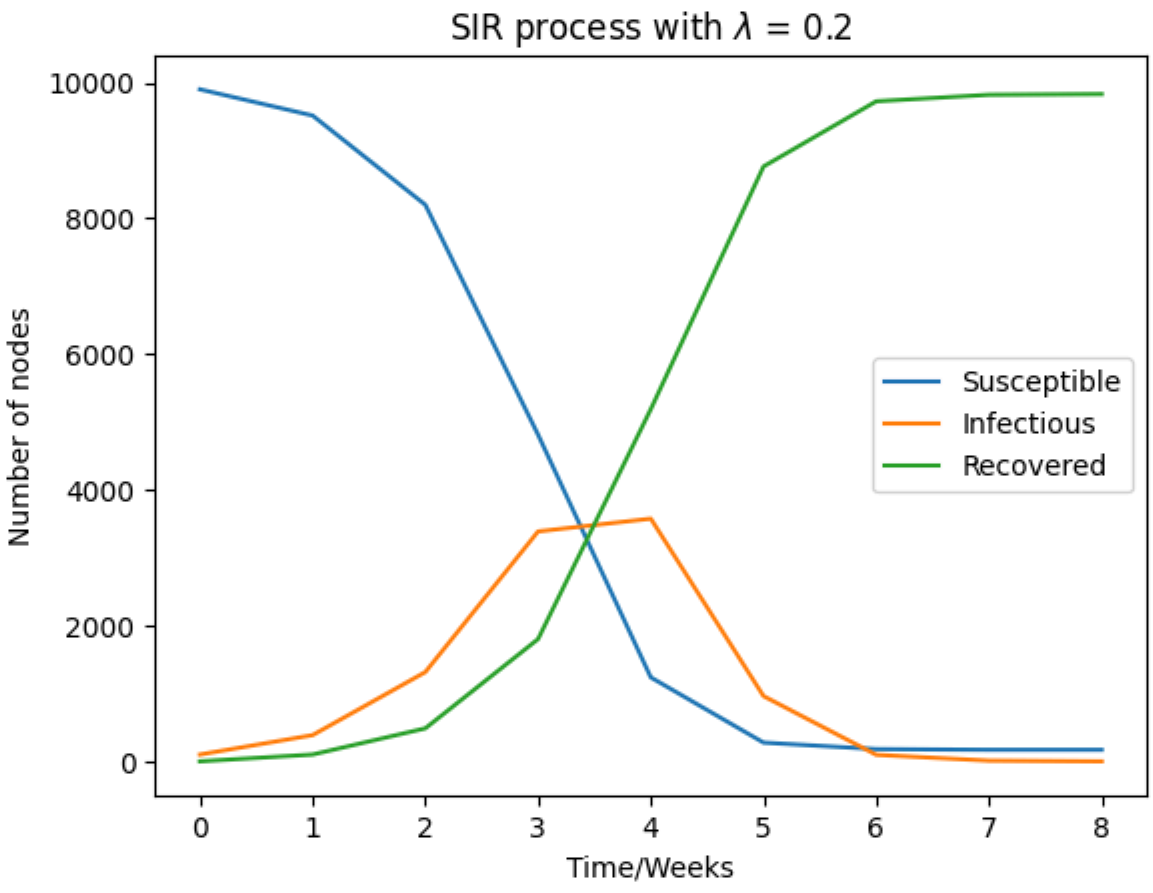
In [ ]:

```
n = 10000
k = 20
p_infected=0.01
p_infect_array = [0.01,0.05,0.1,0.2]
network = Erdos_renyi_Network(n,k)

plt.figure(figsize=(12,8))
for p_infect in p_infect_array:
    simulation = SIR_Model(network,p_infected, p_infect)
    S,I,R = simulation.run_to_extinction()

    plt.plot(S, label='Susceptible')
    plt.plot(I, label='Infectious')
    plt.plot(R, label='Recovered')
    plt.xlabel('Time/Weeks')
    plt.ylabel('Number of nodes')
    plt.title(r'SIR process with $\lambda$ = {}'.format(p_infect))
    plt.legend()
    plt.show()
```






SIR process with λ = 0.1
Susceptible
Infectious
Recovered
Number of nodes
Time/Weeks
0
2000
4000
6000
8000
10000
0
2
4
6
8
10
12
14
16


SIR process with λ = 0.2
Susceptible
Infectious
Recovered
Number of nodes
Time/Weeks
0
2000
4000
6000
8000
10000
0
1
2
3
4
5
6
7
8

Q2

For a range of λ values, inevstigate the number of nodes in state R at the point of extinction. Average over many simulations.

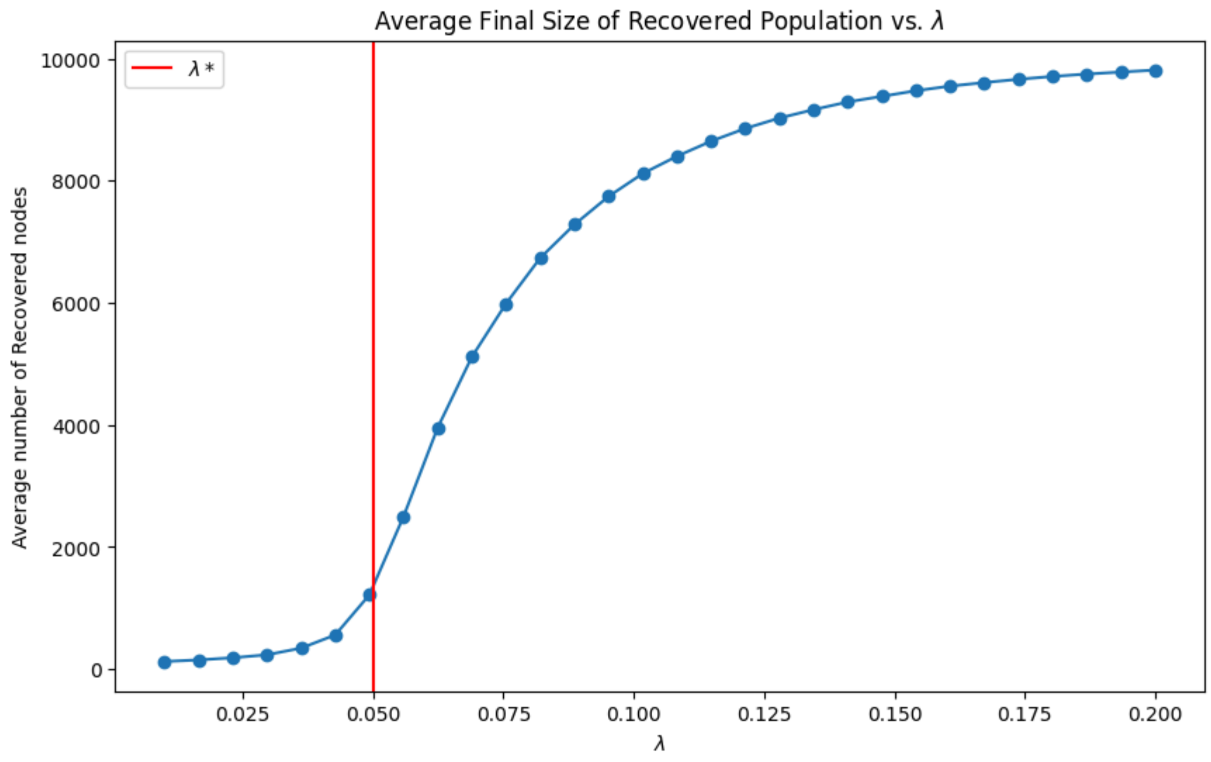
Investigate $\lambda*$, the threshold value when the outbreak goes from only infecting a small fraction to large fraction.

In SIR models, lambda (λ) determines how infectious a disease is. There is a critical value, λc, that separates two qualitatively different dynamical regimes. Here's a basic understanding of those regimes:

Below λc: An outbreak cannot be sustained in the long term. The disease may die out after infecting a small fraction of the population. Above λc: An outbreak can occur and a larger portion of the population will be infected.

In [ ]:
```
def average_R_vs_lambda(network, p_infect_array, p_infected=0.01, runs=100):
    avg_R = []

    for p_infect in p_infect_array:
        R_values = []
        for _ in range(runs):
            simulation = SIR_Model(network,p_infected, p_infect)
            S,I,R = simulation.run_to_extinction()
            R_values.append(R[-1])
        avg_R.append(np.mean(R_values))

    return avg_R

p_infect_array = np.linspace(0.01,0.2,30)
avg_R = average_R_vs_lambda(network, p_infect_array)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(p_infect_array,avg_R, marker='o')
plt.axvline(0.05,label=r'$\lambda*$', color='r')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average number of Recovered nodes')
plt.title(r'Average Final Size of Recovered Population vs. $\lambda$')
plt.show()
```



Q3

Let s_i be the probability that node i is never infected.

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3478503/

$$s_i = \prod_j (1 - \lambda + s_j \lambda)^{A_{ij}}$$

Noting that j are the neighbours of i.

Solve by iteration

1. Fix $s_i \in [0, 1]$ to random values.
2. Iterate the equation until it converges.

Compare the predictions of this equation to simulations.

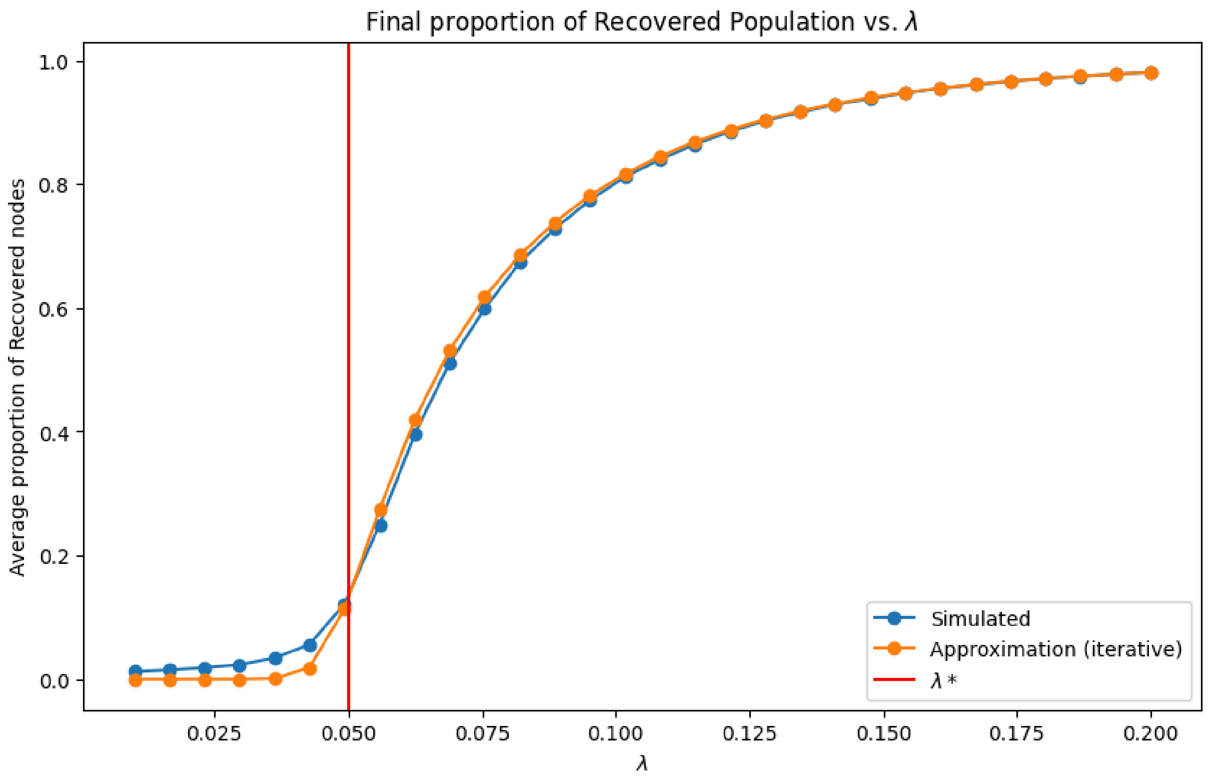
An epidemic Φ is a sequence of states ϕ1, ..., ϕn where ϕi+1 is a possible successor of ϕi for i = 1, ..., n − 1. We will assume that the initial state ϕ1 consists of infectives and susceptibles. The length of this epidemic is ℓ(Φ) = n. Since individuals recover after one step and recovered individuals cannot be reinfected, infection must be transmitted or die out. As a consequence, no epidemic can be longer than the longest self-avoiding path in G = (V, E)/If we assume that each edge transmits or fails to transmit independently, then it is not hard to compute the probability that a susceptible individual is infected by its infected neighbours. This, in turn, allows one to compute the probability that a state ϕ1 is followed by a particular successor state ϕ2. Let us denote this probability by Pr(ϕ2 | ϕ1). This system enjoys the Markov property, that is, the probability of a given state depends only on the previous state. Thus given an initial state Φ1, the probability of the epidemic Φ = ϕ1, ..., ϕn, is

By taking the log of the above equation, relate the stability of $\mathbf{s} = 1$ fixed point to the eigenvalues of A, the adjacent matrix.

Relate this to the simulation and epidemic threshold.

In [ ]:
```
def approx_R_prob_vs_lambda(network, p_infect_array, tol=1e-6, max_iter=20):
    """Returns approximate average probability of final recovered size by iterative
    approx_R = []

    for p_infect in p_infect_array:
        s = np.random.uniform(size=n)

        for _ in range(max_iter):
            s_new = np.ones(network.num_nodes)
            for i in range(n):
                s_new[i]  = np.prod([1+p_infect*(-1+s[j]) for j in network.adj[i]])
            if np.linalg.norm(s_new-s)<tol:
                break
            s = s_new

        approx_R.append(1-np.mean(s))

    return approx_R

approx_R_prob = approx_R_prob_vs_lambda(network,p_infect_array)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(p_infect_array,np.array(avg_R)/network.num_nodes, marker='o',label='Simula
plt.plot(p_infect_array,approx_R_prob, marker='o',label='Approximation (iterative)'
plt.axvline(0.05,label=r'$\lambda*$', color='r')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average proportion of Recovered nodes')
plt.title(r'Final proportion of Recovered Population vs. $\lambda$')
plt.show()
```



The iterative equation and simulation match quite well.

Q4

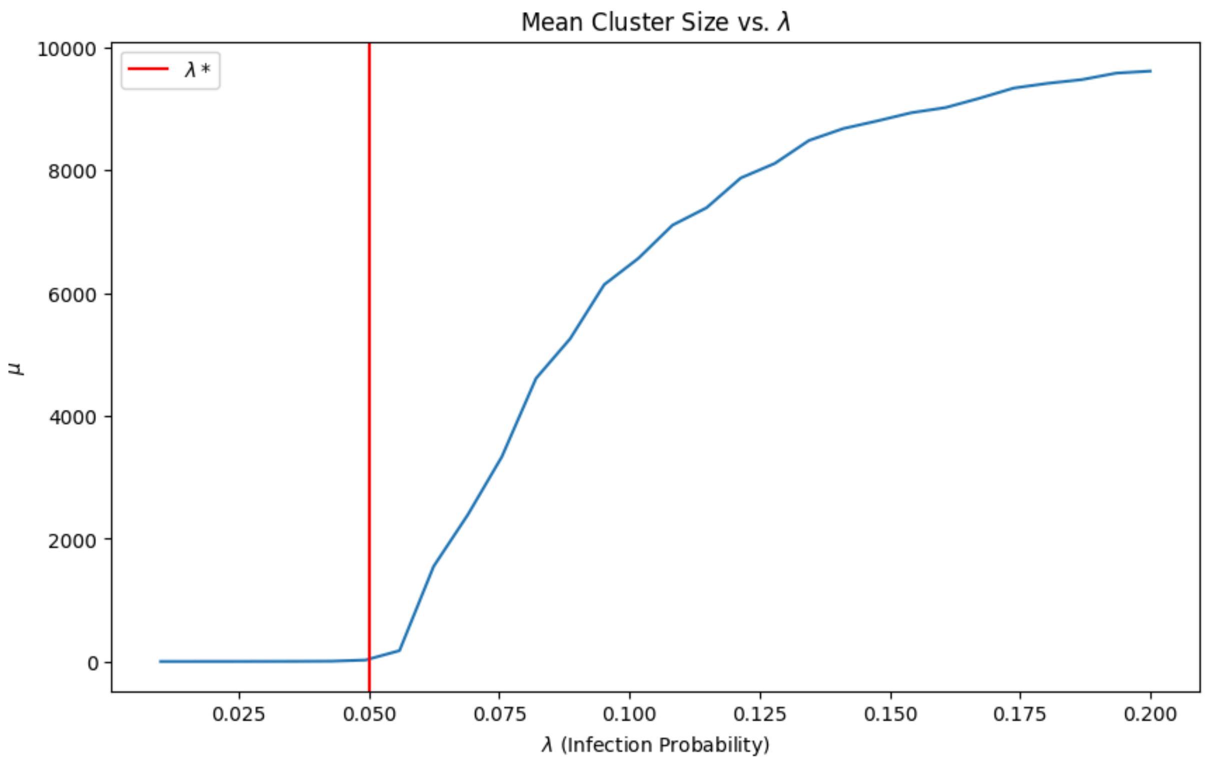
Simulating the whole process can be done more efficiently using Disjoint Set structure.

1. First construct configuration graph as usual.
2. For each edge i,j, merge them with probability λ.
3. Then estimate mean and std deviation of cluster size for the approximate range $\lambda \in (0, 3\lambda_c)$. Where λ_c is the epidemic threshold.

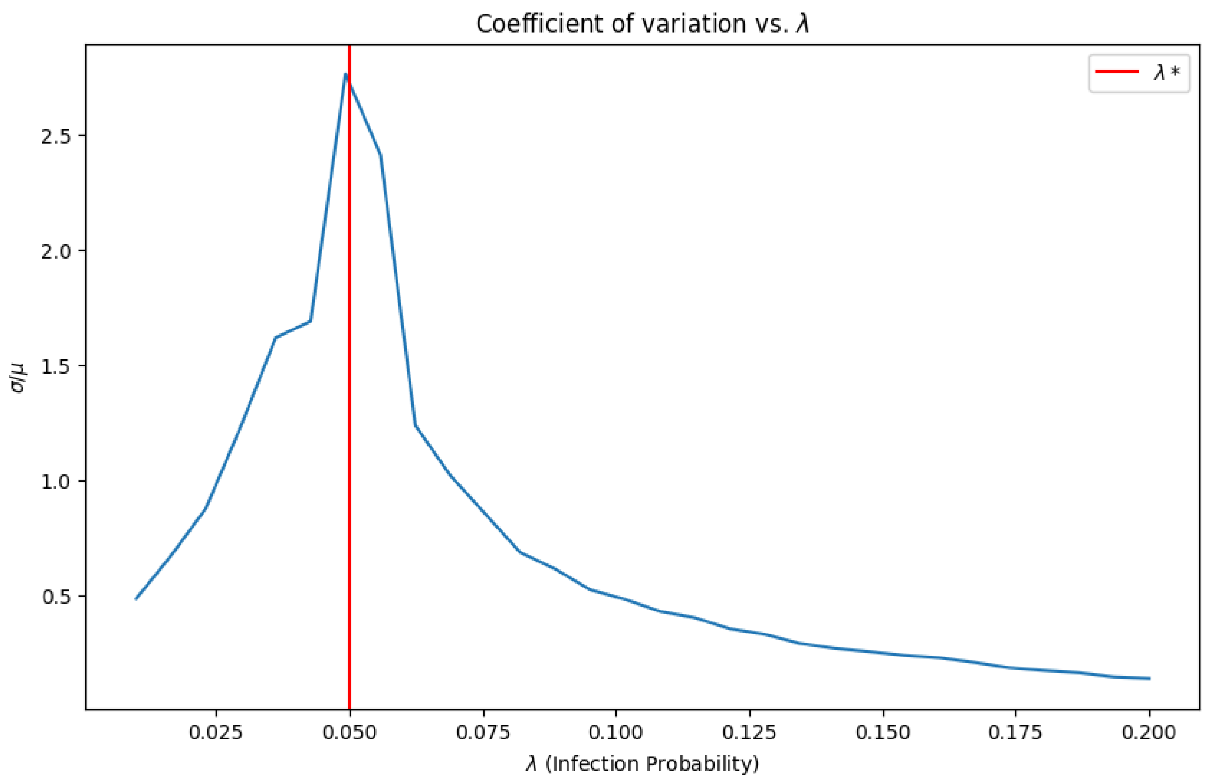
In [ ]:
```
def disjoint_simulation(network, p_infect):
    '''Simulates infection using disjoint set method. Then returns the mean and std

    C = DisjointSet(range(network.num_nodes))
    for i,j in network.edge_list():
        if np.random.random() < p_infect:
            C.merge(i,j)

    cluster_sizes = [len(C.subset(node)) for node in C]

    return np.mean(cluster_sizes),np.std(cluster_sizes)

mean_cluster_sizes = []
std_cluster_sizes = []

P_infect_array = np.linspace(0,0.05*3,20)
```

```
for p_infect in p_infect_array:
    cluster_mean,cluster_std = disjoint_simulation(network,p_infect)
    mean_cluster_sizes.append(cluster_mean)
    std_cluster_sizes.append(cluster_std)
```

In [ ]:
```
# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(p_infect_array, mean_cluster_sizes)
plt.axvline(0.05,color='r',label=r'$\lambda*$')
plt.xlabel(r'$\lambda$ (Infection Probability)')
plt.ylabel(r'$\mu$')
plt.title(r'Mean Cluster Size vs. $\lambda$')
plt.legend()
plt.show()
```



In [ ]:
```
# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(p_infect_array, np.array(std_cluster_sizes)/np.array(mean_cluster_sizes))
plt.axvline(0.05,color='r',label=r'$\lambda*$')
plt.xlabel(r'$\lambda$ (Infection Probability)')
plt.ylabel(r'$\sigma / \mu$')
plt.title(r'Coefficient of variation vs. $\lambda$')
plt.legend()
plt.show()
```



The coefficient of variation (σ/μ) is a measure of variability relative to the mean. In the context of the SIR model, it might represent the variability in the size of clusters of infected individuals.

Around $\lambda*$, the coefficient of variations starts to increase significantly/rapidly.

Q5 Even more efficient simulation

This is an efficent way of determining cluster sizes as a function of λ via one sweep through all edges.

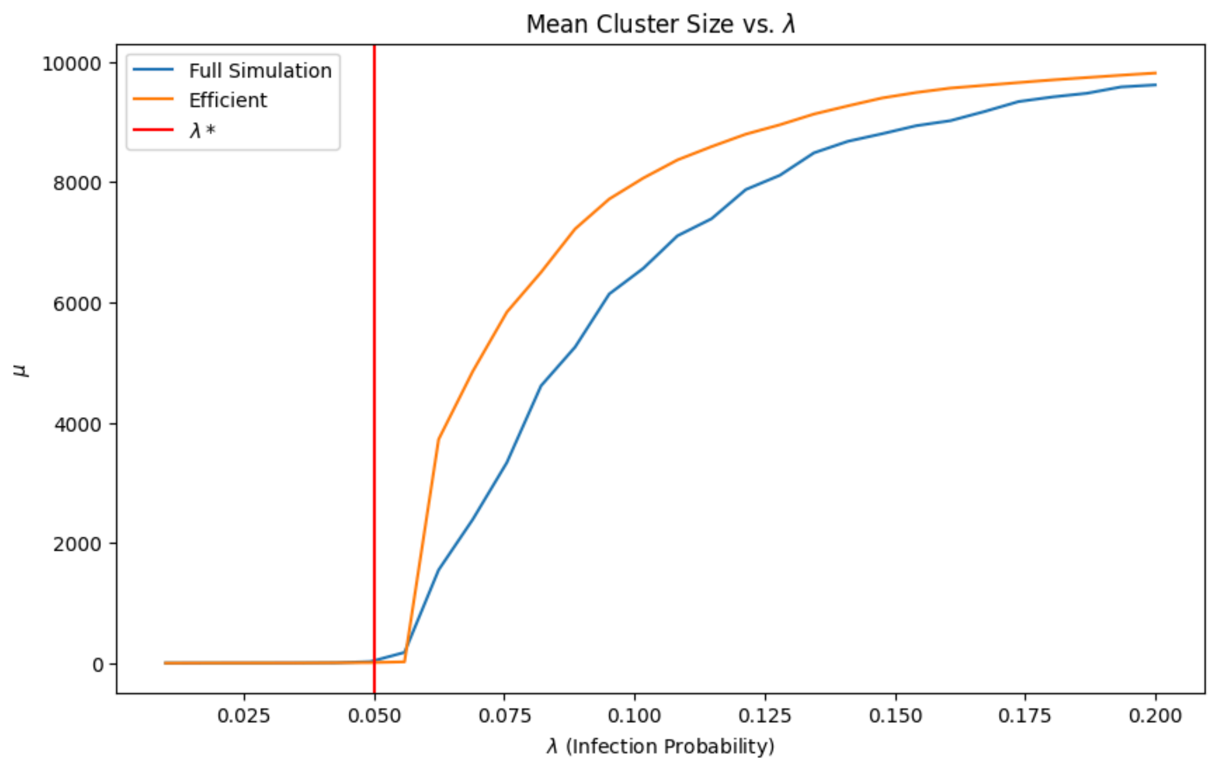
1. Generate a configuration network as usual
2. Let k be the current number of edges selected/worked through.
3. Compute average cluster size at each $k \in [1, m]$.
4. Then compute average cluster size for a given λ using binomial distribution with argument k and total size m.

```
In [ ]: def efficient_disjoint_simulation(network):
            '''Efficiently simulates infection using disjoint set method. Then returns aver

            cluster_sizes_vs_k = []
            edges = network.edge_list()

            random.shuffle(edges)
            C = DisjointSet(range(network.num_nodes))
```

```
    for k in range(network.num_edge):
        i,j = edges[k]
        C.merge(i,j)

        cluster_sizes_vs_k.append(len(C.subset(0)))

    return cluster_sizes_vs_k

cluster_sizes_vs_k = efficient_disjoint_simulation(network)
```

In [ ]:
```
mean_cluster_sizes_efficient = []

for p_infect in p_infect_array:
    probability_array = scistats.binom.pmf([k for k in range(1,network.num_edge+1)]
    mean_cluster_sizes_efficient.append(np.sum(np.array(cluster_sizes_vs_k)*np.arra
```

In [ ]:
```
# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(p_infect_array, mean_cluster_sizes,label='Full Simulation')
plt.plot(p_infect_array, mean_cluster_sizes_efficient,label='Efficient')
plt.axvline(0.05,color='r',label=r'$\lambda*$')
plt.xlabel(r'$\lambda$ (Infection Probability)')
plt.ylabel(r'$\mu$')
plt.title(r'Mean Cluster Size vs. $\lambda$')
plt.legend()
plt.show()
```



The efficient disjoint simulation took about 5 mins and the simple disjoint simulation took ~10 mins, due to the need of computing $c_i(k)$ for $k \in [1, m]$ for the efficient case, where m is a large number.But after having $c_i(k)$ once, it is much faster to compute $c_i(\lambda)$ for any $\lambda$.