PSTAT 160B Final Project: Graph Theory Applications and Monte Carlo Simulation

Hailey Broderick, Elizabeth Forney, Vasyl Ostapenko

Fall 2021

1 Introduction

In this paper we analyze the mathematics appearing in the movie *Good Will Hunting*. Concretely, after introducing and discussing concepts from graph theory, stochastic process, and probability we will use them to solve the first two parts of the central problem in the film. Afterward, we discuss the graph theory concept of cover times and implement Monte Carlo simulation methods in Python to study the cover times of various graphs. All throughout, we take a probabilistic perspective as opposed to the traditional mathematical view as we tackle the problems.

2 Graph Theory

Graph Theory is the study of graphs, which are structures that mathematically model relationships between networks of points (or vertices) and lines (or edges). We can define a **graph** $G(\mathbb{V}, \mathbb{E})$ as a collection of vertices, $\mathbb{V}$, connected by edges, $\mathbb{E}$. An edge, $\{i, j\} \in \mathbb{E}$, represents a connection between the vertices $i, j \in \mathbb{V}$. We can also classify some graphs as either directed or undirected and/or simple. A graph is **simple** if for all $i \in \mathbb{V}$, there are no edges $\{i, i\} \in \mathbb{E}$ (i.e. no self-loops) and there is at most one edge between any two vertices. For example, a simple graph with three vertices ($\mathbb{V} = \{1, 2, 3\}$) could look like the following:

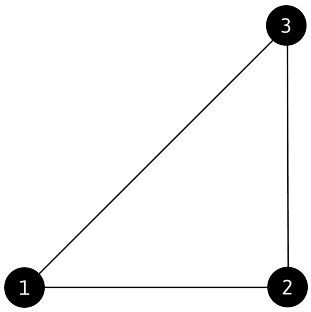

Figure 1: Example of a simple graph

In figure 1, the edges are $\mathbb{E} = \{(1,2),(1,3),(2,3)\}$. There is no more than one edge between any two vertices.

Moreover, a graph is called **undirected** if all edges are undirected. An undirected edge means that the edge has no direction and thus an edge $(i,j) \in \mathbb{E}$ is the same as an edge $(j,i) \in \mathbb{E}$. Visually, an example of an undirected graph with 5 vertices can look like the figure below.

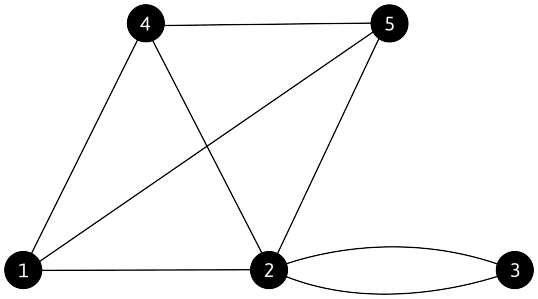

Figure 2: Example of an undirected graph

In our example of an undirected graph above, we can identify the set of vertices $\mathbb{V} = \{1,2,3,4,5\}$ and set of edges $\mathbb{E} = \{(1,2),(1,4),(1,5),(2,4),(2,5),(2,3),(2,3),(4,5)\}$.

Conversely, a graph is called **directed** if all edges are directed. A directed edge is an edge that has direction so that $(i,j) \in \mathbb{E}$ is *not* the same as an edge $(j,i) \in \mathbb{E}$. Below is an example of a directed graph with 4 vertices.

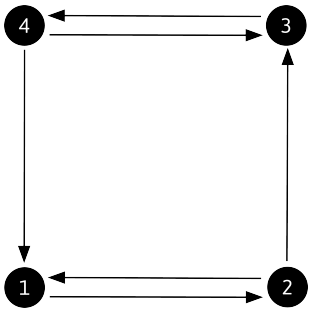

Figure 3: Example of a directed graph

The directed graph above has vertices $\mathbb{V} = \{1, 2, 3, 4\}$ with directed edges $\mathbb{E} = \{\{1, 2\}, \{2, 1\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{4, 3\}\}$. Additionally, it is important to note that a graph can also have both directed and undirected edges.

3 Adjacency Matrix

Suppose we have a graph $G(\mathbb{V}, \mathbb{E})$, either directed or undirected. We can define its **adjacency matrix**, A, as a $|\mathbb{V}| \times |\mathbb{V}|$ matrix where each entry A_{ij} is the number of edges from vertex $i \in \mathbb{V}$ to vertex $j \in \mathbb{V}$. For a directed edge from vertex i to vertex j, the edge (j, i) is not counted.

In graph theory, an adjacency matrix A describes the structure of a (finite) graph. The dimensions of matrix A are dictated by the number of vertices $n \in \mathbb{N}$ in the graph (so, $n = |\mathbb{V}|$). The entries of matrix, A_{ij} describe every possible connection (signified by an edge) between each vertice $i \in \mathbb{V}$ (row index) and $j \in \mathbb{V}$ (column index).

4 Good Will Hunting Problem

Equipped with definitions from graph theory and adjacency matrices, we can now examine a related problem that appears in the movie *Good Will Hunting*. Below is a part of the problem set that appeared in this movie.

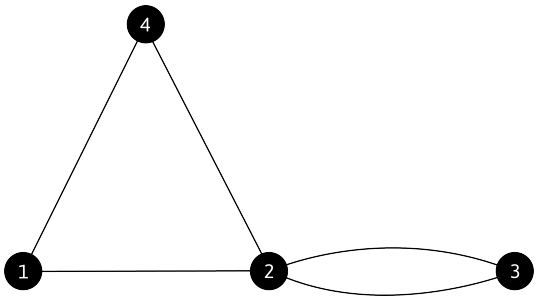

Figure 4: Graph of G

1. Find the adjacency matrix, A.
2. Find the matrix giving the number of 3 step walks.

The *Good Will Hunting* movie also displays the following solutions to the two questions posed.

Solution 1.

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Solution 2.

$$A^3 = \begin{pmatrix} 2 & 7 & 2 & 3 \\ 7 & 2 & 12 & 7 \\ 2 & 12 & 0 & 2 \\ 3 & 7 & 2 & 2 \end{pmatrix}$$

In this section we will explain the solutions using graph theory and apply the definition of the adjacency matrix.

From figure 1 we can identify the vertices $\mathbb{V} = \{1, 2, 3, 4\}$ and the edges $\mathbb{E} = \{(1, 2), (1, 4), (2, 3), (2, 3), (2, 4)\}$. It is important to note that $(2, 3)$ is a double edge, meaning that there are two distinct undirected edges between vertex 2 and vertex 3. By the definition of adjacency matrix, as defined in section 3, the solution for question 1 is trivial. Each entry A_{ij} is the number of edges from vertex i to vertex j. For example, entry $A_{2,3} = 2$ means that there are two edges from vertex 2 to vertex 3, which is confirmed by our identification of the set of edges $\mathbb{E}$.

For the second Good Will Hunting problem, the matrix that gives the number of 3 step walks is the matrix A^3. This is because A^k_{ij} is the number of ways to walk from vertex i to vertex j in k steps which is shown through the induction proof below:

Proof. (by induction).

For $k = 1 : (A^k)_{ij} = A_{ij}$ because by the definition of an adjacency matrix, A_{ij} is the number of ways to walk from vertex i to vertex j in 1 step.

Suppose $(A^{k-1})_{ij}$ is the number of ways to walk from vertex i to vertex j in $k-1$ steps. Then,

For k:

$$\begin{aligned}(A^k)_{ij} &= \left(\prod_{i=1}^{k} A\right)_{ij} \\ &= \left(A \cdot \prod_{i=1}^{k-1} A\right)_{ij} \\ &= (A \cdot A^{k-1})_{ij} \\ &= \sum_{\ell=1}^{n} (A^{k-1})_{i\ell} A_{\ell j} \\ &= (A^{k-1})_{i1}A_{1j} + (A^{k-1})_{i2}A_{2j} + \ldots + (A_{in}^{k-1})A_{nj}\end{aligned}$$

Thus, $(A^k)_{ij}$ is the sum of the number of ways to walk from vertex i to vertex ℓ in $k-1$ steps times the number of ways to walk from vertex ℓ to vertex j in 1 step $\forall \ell \in \mathbb{S}$. Therefore, $(A^k)_{ij}$ is the number of ways to walk from vertex i to vertex j in k steps. $\square$

Hence, the matrix A^3 is the the matrix giving the number of 3 step walks since each entry A^3_{ij} is the number of ways to walk from vertex i to vertex j in 3 steps. We can compute the matrix A^3 via matrix multiplication as follows:

$$\begin{aligned}A^3 &= A \cdot A \cdot A \\ &= \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 1 & 2 & 1 \\ 1 & 6 & 0 & 1 \\ 2 & 0 & 4 & 2 \\ 1 & 1 & 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 7 & 2 & 3 \\ 7 & 2 & 12 & 7 \\ 2 & 12 & 0 & 2 \\ 3 & 7 & 2 & 2 \end{pmatrix}\end{aligned}$$

5 Random Walk on Graphs

In order to define a random walk on a graph, we must first describe the stochastic process underlying the walk itself.

Generally, A **stochastic process**, $X = \{X_k\}$, is a collection of random variables, usually indexed by time k. The **state space**, $\mathbb{S}$, of such a process

consists of all the possible states which the process X may take on (i.e $X_k \in \mathbb{S}$ $\forall k$). The random variable $X_k \in \mathbb{S}$ is the state/value of the stochastic process at time k ($k \in \mathbb{N} \implies$ discrete time).

A stochastic process X is called a **Markov Chain (MC)** when the conditional probability of the process X reaching future state X_{k+1}, given the past and states $X_0, ..., X_{k-1}$ and the present state, X_k, only depends on the present state.

A special type of MC , a **Random Walk (RW)**, is a process where at each time step k, the MC randomly walks from some $i \in \mathbb{S}$ to a neighboring state $j \in \mathbb{S}$ with some probability P_{ij}. The transition probabilities of the RW can be stored in a transition matrix $\mathbf{P}$ which is a $|\mathbb{S}| \times |\mathbb{S}|$ matrix with entries s.t. $P_{ij} = P(X_{k+1} = j | X_k = i)$. In order to be a valid transition probability matrix, $\mathbf{P}$, must satisfy two properties:

1. For each entry of $\mathbf{P}, P_{ij} \in [0, 1]$
2. Every row of $\mathbf{P}$ sums to 1. (i.e. for every $i \in \mathbb{S}, \sum_{\forall j \in \mathbb{S}} P_{ij} = 1$)

A **Random Walk on a Graph** $G(\mathbb{V}, \mathbb{E})$ is a random sequence of vertices, v_i's, where there exists an edge from v_i to v_{i+1}. Therefore, at each time step, the walk must follow the direction of directed edges, or can traverse in either direction for undirected edges. Thus, the state space of a random walk on a graph is equivalent to the vertices of the graph, $\mathbb{S} = \mathbb{V}$. At each time step, the random walk can move from any vertex $i \in \mathbb{V}$ to another vertex $j \in \mathbb{V}$ with some probability P_{ij}, so long as there exists an edge between the vertices, i.e $\exists$ $(i, j) \in \mathbb{E}$.

Now we will consider a Random Walk on the Good Will Hunting graph G.

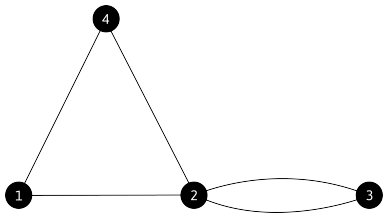

Figure 5: Graph of G

Let $X = \{X_k\}$ for $k \in \mathbb{N}$ be a MC representing a random walk on (undirected) graph G. Recall that the state space of the RW is $\mathbb{S} = \mathbb{V} = \{1, 2, 3, 4\}$ and that the graph G has edges $\mathbb{E} = \{(1, 2), (1, 4), (2, 3), (2, 3), (2, 4)\}$.

Recall that the entries of Adjacency matrix A, A_{ij}, indicate the number of ways to walk between any two vertices $i, j \in \mathbb{V}$ (i.e. the number of edges

between i, j). The adjacency matrix A of Good Will Hunting Graph G:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

We are able to derive the transition probability matrix of graph G from adjacency matrix A.

First, create a diagonal matrix D where the ith diagonal entry equals the sum of the ith row entries of matrix A.
(i.e. for each $i \in \mathbb{V}, D_{ii} = \sum_{\forall j \in \mathbb{V}} A_{ij}$):

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

Second, compute the inverse of diagonal matrix D:

$$D^{-1} = \begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/4 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix}$$

Lastly, compute the (one step) transition probability matrix of a RW on graph G by multiplying the adjacency matrix A by the inverse diagonal matrix D^{-1}:

$$\implies \mathbf{P} = D^{-1}A = \begin{pmatrix} 0 & 1/2 & 0 & 1/2 \\ 1/4 & 0 & 1/2 & 1/4 \\ 0 & 1 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \end{pmatrix}$$

This matrix is a valid transition probability matrix for the RW X on graph G, since the two properties for a well-defined transition probability matrix hold. First, each entry of $\mathbf{P}, P_{ij}$, is in the interval $[0, 1]$ and second, for every $i \in \mathbb{S}, \sum_{\forall j \in \mathbb{S}} P_{ij} = 1$. Note: P is a $|\mathbb{V}|$x$|\mathbb{V}|$ matrix (i.e a 4x4 matrix), where each entry P_{ij} represents the probability of the RW moving from vertex i to vertex j in one step.

Adjacency matrices and the corresponding transition probability matrices play a key role in examining and predicting the behavior of Random walks on graphs. As we saw in our Good Will Hunting problem, graph theory allows us to construct adjacency matrices for graphs. These adjacency matrices describe the possible movements of a RW (stochastic process) on a graph. Based on these adjacency matrices and the Markov Property of a RW on a graph, we are able to derive the transition probabilities of a random walk on a graph.

We will now examine six well known graphs on which random walks may occur.

The first graph we will define is what is known as a **Complete graph**. This is a simple undirected graph where there is an undirected edge between between every vertex $i \in \mathbb{V}$ and vertex $j \in \mathbb{V}$ such that $i \neq j$. Below is what a complete graph with 4 vertices looks like.

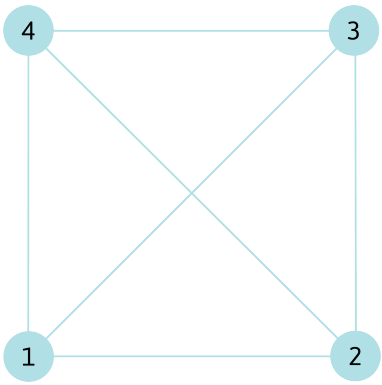

Figure 6: Complete graph with 4 vertices.

Next, we will define a **Lollipop graph**. This graph consists of a complete graph with a path graph connected to one of the vertices. A path graph is an unidrected graph where the vertices are ordered and there is an undirected edge between each vertex $v_i \in \mathbb{V}$ and $v_{i+1} \in \mathbb{V}$. Below is an example of what a lollipop graph with 4 vertices looks like.

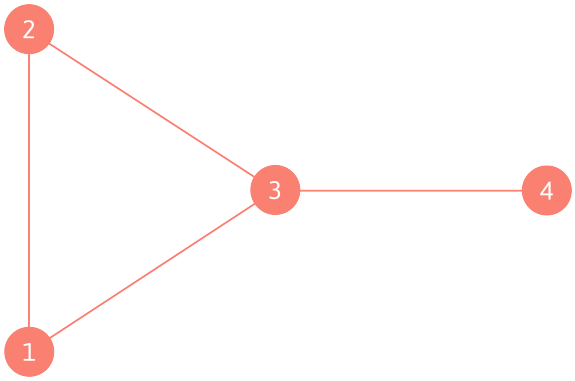

Figure 7: Lollipop graph with 4 vertices.

Furthermore, there is a graph called the **Star graph** that is defined by a central vertex $i \in \mathbb{V}$ where there is an undirected edge between it and every

vertex $j \in \mathbb{V}$ such that $i \neq j$. Below is what a star graph with 4 vertices looks like.

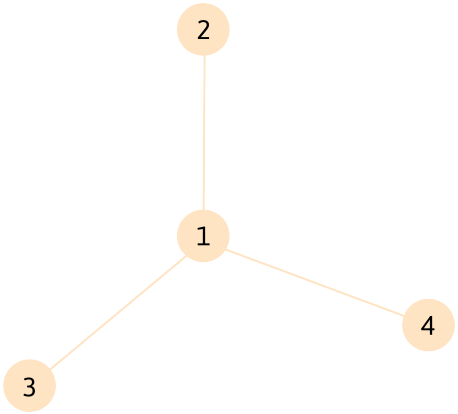

Figure 8: Star graph with 4 vertices.

We will now define a **Tree graph**. This is an undirected graph such that there is exactly one undirected edge between any two vertices $i \in \mathbb{V}$ and $j \in \mathbb{V}$. An example of a tree graph with 4 vertices looks like:

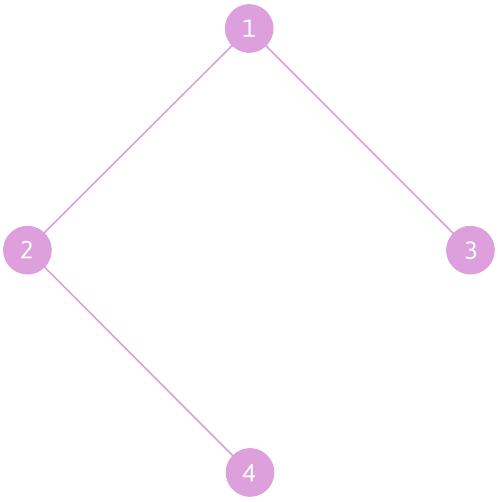

Figure 9: Tree graph with 4 vertices.

Moreover, another graph which a random walk can be on is a **Regular graph**. A regular graph is a graph with n vertices, in which each vertex has the same pre-defined d-number of edges. The figure below is an example of a regular graph with four vertices, each with $d = 2$ undirected edges.

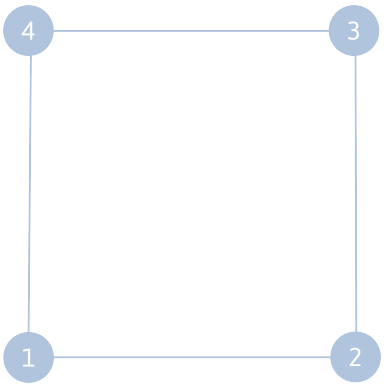

Figure 10: Regular graph with 4 vertices.

The final graph we will discuss is the **Random graph**. In our simulation experiment in section 7 below, we define this graph based on the Erdős-Rényi model for generating random graphs. Beginning with n vertices where each vertex i has exactly one edge with another vertex j ($i \neq j$), we choose a random x-number of additional edges for each vertex to have such that $x \in \{0, 1, ..., n-1\}$. Below is an example of one instance of a random graph with four vertices.

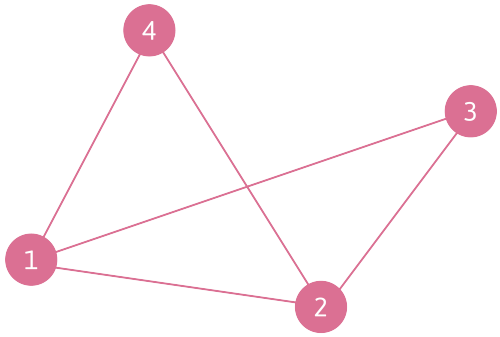

Figure 11: Random graph with 4 vertices.

6 Cover Times

Suppose we have a Markov Chain with a state space $\mathbb{S}$ that has n number of states. The **cover time** can be defined as the random time C taken for all the states in $\mathbb{S}$ to be visited. Mathematically, if we let $n = |\mathbb{V}|$ and $\mathcal{D}_k$ be the number of distinct vertices visited at time $k \in \mathbb{N}$, then we can define the cover time for a graph as

$$C_n = \min\{k \geq 0 : \mathcal{D}_k = n\}$$

Thus, the cover time C_n is the minimum amount of time (k time steps) taken

for a RW on a graph to hit each distinct vertex (n distinct vertices total) at least once. The concept of cover times is important because we would like to examine cover times for several different graphs (simulation experiment discussed in section 7) to see what properties of a graph contribute to a longer/shorter cover time.

7 Simulation Experiments

In this section we will discuss our simulation of random walks on the Good Will Hunting graph G, as well as our simulation experiment on cover times for random walks on various different types of graphs.

7.1 Random Walks on Good Will Hunting Graph

Our first simulation consists of simulating several random walks on the Good Will Hunting graph, G, presented in section 4. Each random walk we simulated comprises of 50 discrete time-steps and the vertex at time-step 0 is chosen uniformly at random. We would like to show that there are many possible different random walks that can occur on the same graph with the same transition probabilities. Below are the visual representations of our three simulations of a 50-step random walk on the Good Will Hunting graph.

Figure 12: Random walk #1 on Good Will Hunting graph

An animated visual of the random walk shown in figure 12 can be found here.

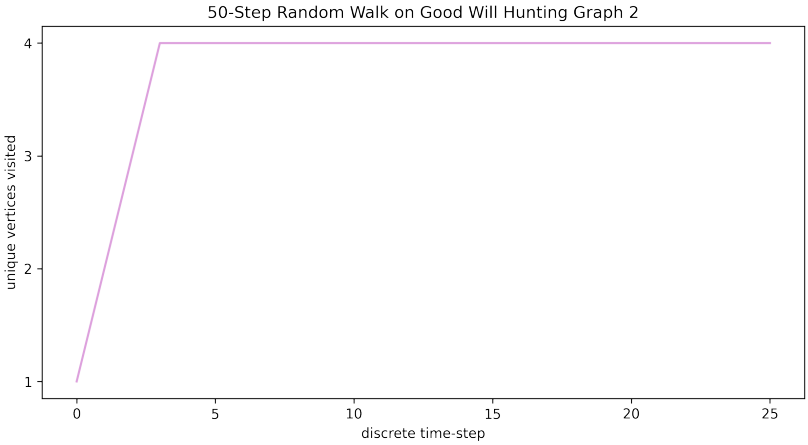

Figure 13: Random walk #2 on Good Will Hunting graph

An animated visual of the random walk shown in figure 13 can be found here.

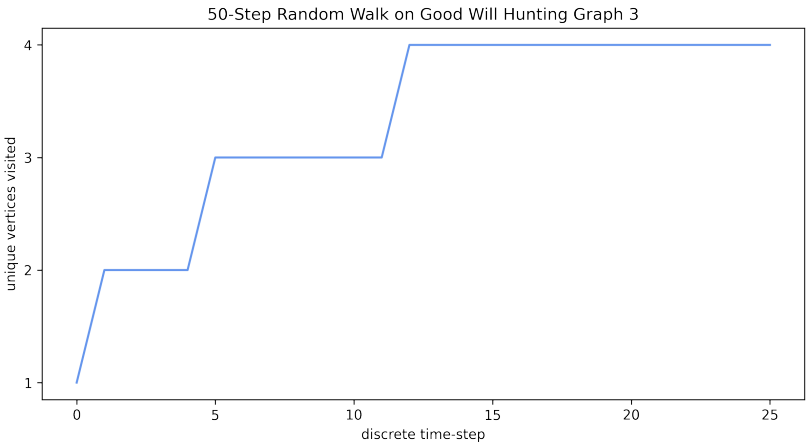

Figure 14: Random walk #3 on Good Will Hunting graph

An animated visual of the random walk shown in figure 14 can be found here.

As we can see in the three graphs above, none are a replica of another even though they are walks on the same graph with the same transition probabilities. The cover time, C_n, for each of these three random walks are also different. The cover times for figures 12, 13, and 14 are $6, 4,$ and 12 respectively. This further exemplifies the randomness of the walks.

7.2 Cover Times For Various Graphs

Now, we will discuss our simulation experiment of expected cover times for the following graphs: Good Will Hunting graph, Good Will Hunting graph starting in the stationary distribution, Complete graph, Lollipop graph, Star graph, Tree graph, Regular graph, and Random graph.

I. Purpose The purpose of our graph cover time experiments is to statistically determine, via Monte Carlo simulation, which graph out of a curated selection has the lowest and highest expected cover time respectively. In doing so, we are showing how graph structures (and stochastic processes i.e. RW's on these graphs) can model mathematical scenarios and determine desired information. It has been conjectured that the Complete graph (fully-connected graph) minimizes cover time, while the Tree graph maximizes it, but this has never been proven. We take the stochastic processes approach in giving support to the conjectures.

II. Hypothesis Our hypothesis is that the Complete graph will minimize cover time, while the Tree graph will maximize cover time. This hypothesis is based on intuition and graph theory. The fully-connected graph structure gives a random process the chance to reach any node at every step. Thus we would expect the random walk to reach all nodes quicker than some graph structures which limit node access. Meanwhile, the Tree graph disallows a node from having more than one edge and thus limits node access the most at each step. Therefore we would expect the cover time for a Tree graph to be the highest.

III. Method We used Monte Carlo simulation to generate NSIM=2500 random walks on eight different graphs each. The cover time per walk per graph was tracked and thus we were able to compute the expected (average) cover time per graph. The start node distribution for each walk was random uniform for all graphs except "GWH pi" graph, with that one starting from a random node choice from the stationary distribution of the Good Will Hunting graph. Every graph was limited to four nodes to enable direct comparison to the Good Will Hunting Graph. Moreover, functions implemented in the networkx Python package for graphs were used to generate each of the complete, lollipop, star, tree, regular, and fully-random graphs.

IV. Results For each type of graph, we plotted a histogram of the cover time for each of the 2500 random walk simulations. After, in order to compare the cover times, we created a bar plot of the expected cover times for each type of graph. These histograms and bar plot are presented below.

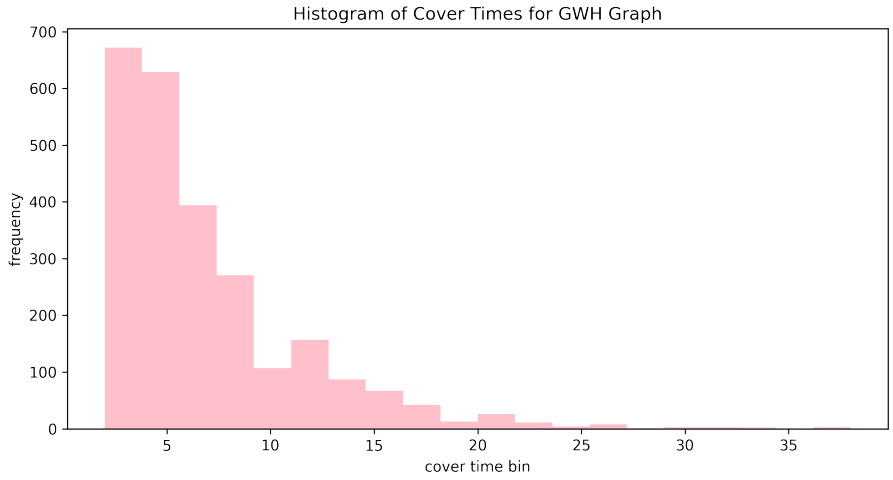

Figure 15: Histogram of cover times for the Good Will Hunting graph

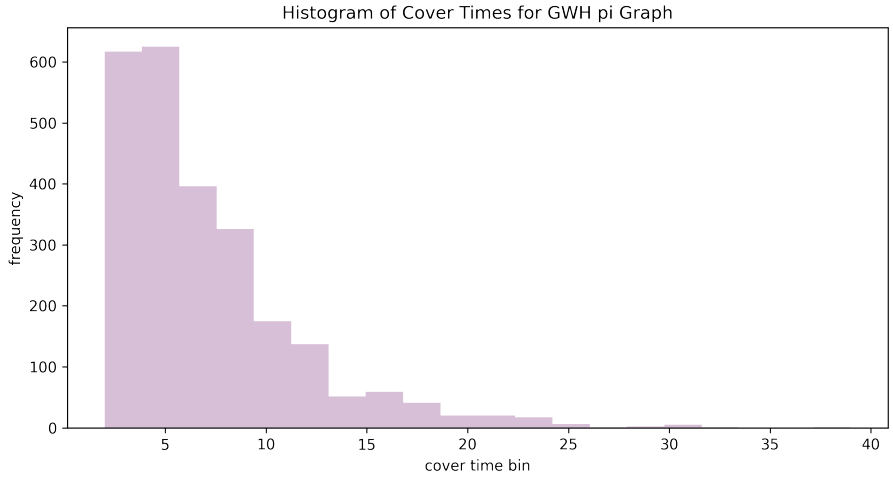

Figure 16: Histogram of cover times for the Good Will Hunting graph starting in the stationary distribution

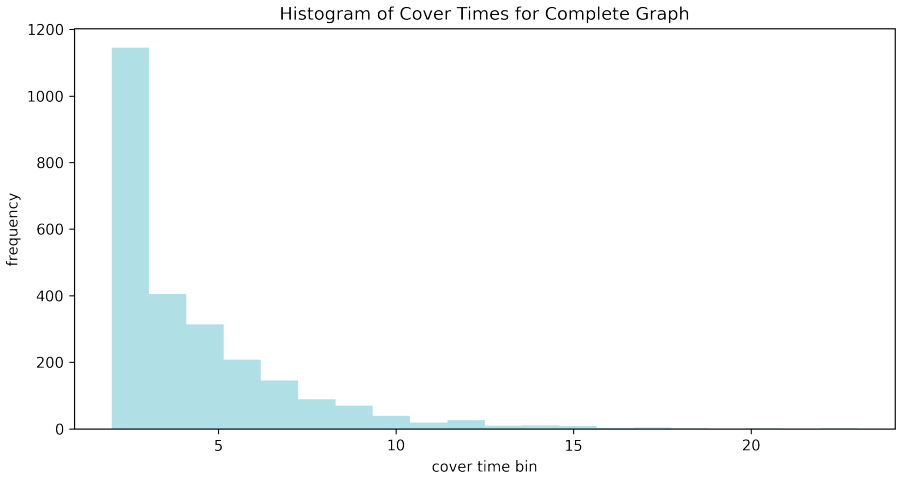

Figure 17: Histogram of cover times for the Complete graph

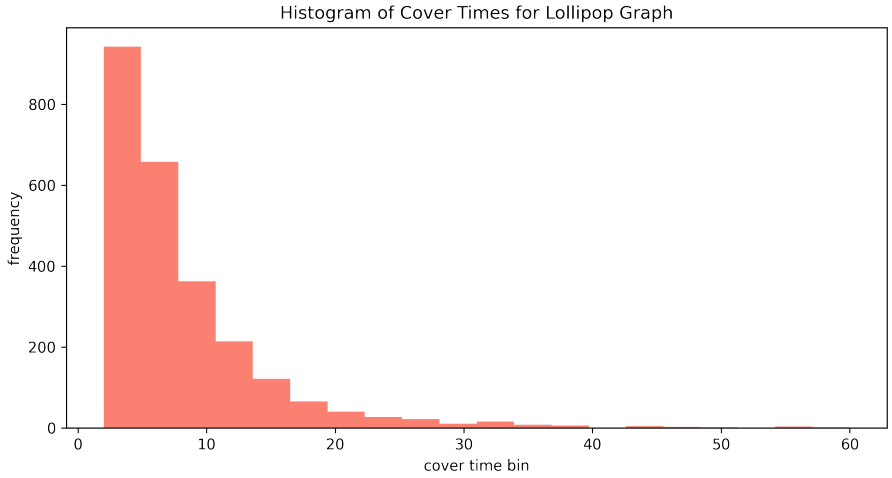

Figure 18: Histogram of cover times for the Lollipop graph

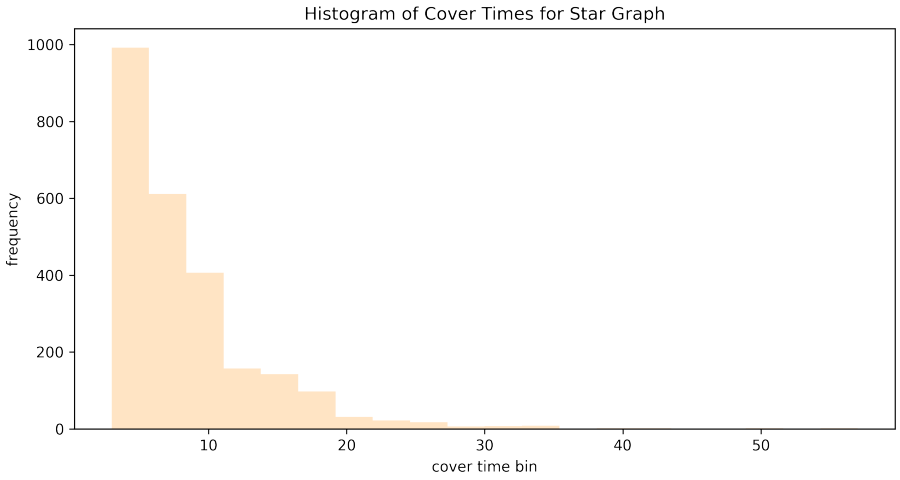

Figure 19: Histogram of cover times for the Star graph

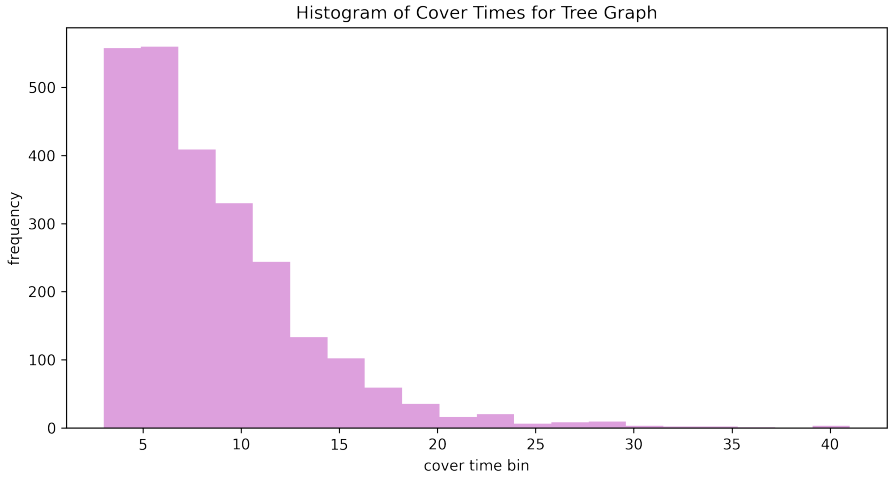

Figure 20: Histogram of cover times for the Tree graph

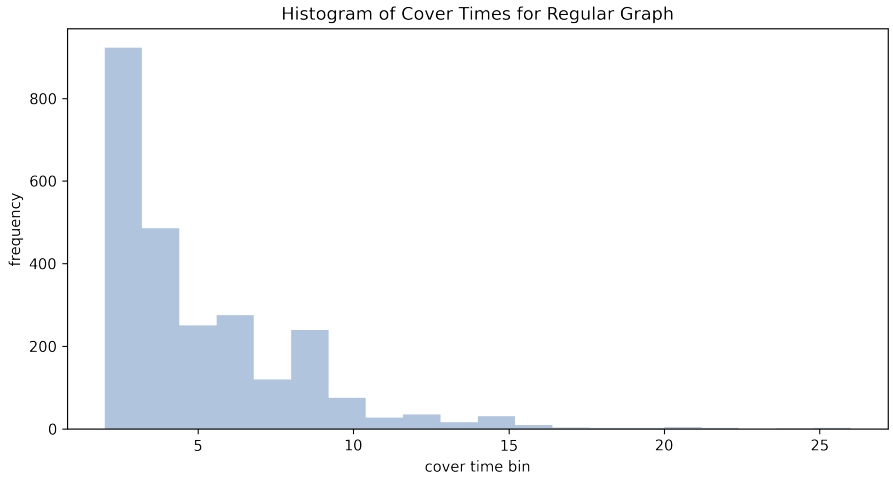

Figure 21: Histogram of cover times for the Regular graph

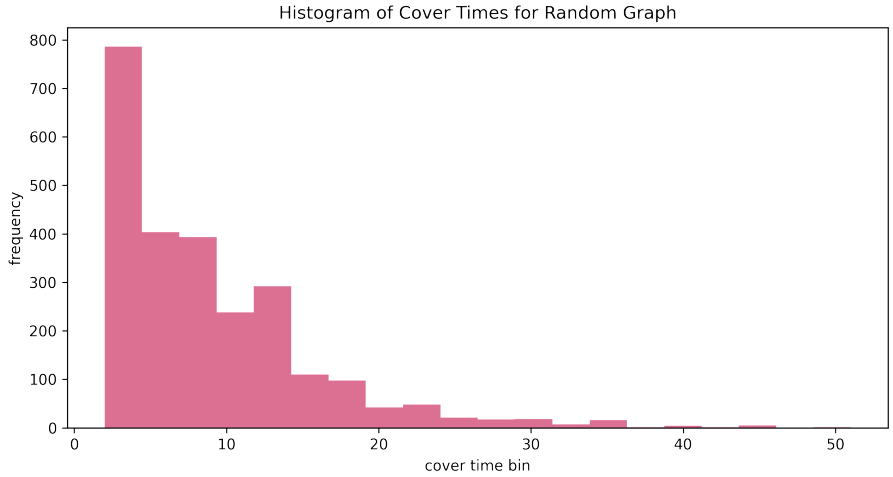

Figure 22: Histogram of cover times for the Random graph

As seen in the histograms above, some types of graphs have a larger range of cover times while other have shorter ranges. Additionally, some have cover times that are tightly concentrated to the left while others have a set of cover times that are more spread out. For example, the Tree graph has a large range of cover time values (from 4 to 40) and the shape of the histogram tails off slowly. Conversely, the histogram for the Complete graph has a short range of values (4

through 20) and the values are largely concentrated around the smallest values. These histograms support the bar plot of expected cover times for each graph below because, the tightly concentrated graphs with shorter ranges tend to have a smaller expected cover time compared to graphs where the cover times are distributed across a larger range.

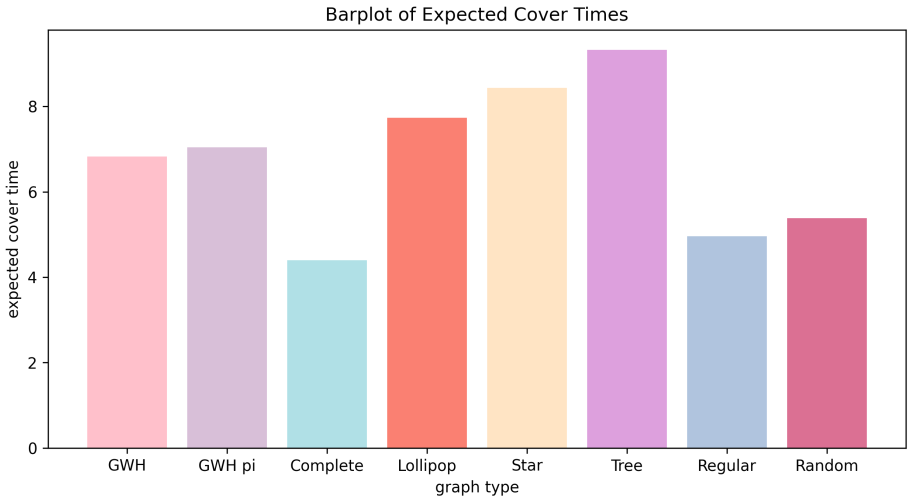

Figure 23: Bar plot of expected cover times for each type of graph

The resulting bar plot of cover times, shown above, gave evidence in support of our initial conjecture. Monte Carlo simulation of random walks on different graphs points to the fact that the fully-connected (Complete) graph may minimize cover time while the Tree graph may maximize it. It is interesting to note that, with regard to the Good Will Hunting graph, picking a start node from the stationary distribution results in a longer expected cover time compared to when a start node is picked from the uniform random distribution.

V. Analysis Picking a node from the stationary distribution of the Good Will Hunting graph results in the highest probability being assigned to the central node 2. Thus, the resulting longer cover time may be explained by the graph structure - the random walk may get stuck in a loop between nodes 2 and 3 and thus fail to reach nodes 1 and 4 for a long time. Meanwhile, the uniform random distribution gives higher weights to nodes 1, 3, and 4, meaning the walk may be more efficient in reaching all nodes.

VI. Conclusion Monte Carlo simulations are a powerful tool given the rapid rise in computing power in recent times. Given a deterministic problem, we could easily derive a good approximation to the problem by taking a random processes approach. Graph theory allows us to structure mathematical scenarios and simulate stochastic processes on these graphs. A computer can use random

sampling and repeatedly generate the outcome of an experiment, all of which could then be averaged to arrive at a close approximation of the real value.

8 References

[1] Learning material comes from UCSB PSTAT 160B lectures and homework by Professor Alex Shkolnik.

[2] Aldous, D., Fill, J. A. (2002). *Reversible Markov Chains and Random Walks on Graphs*. 207-233.

[3] Complete graph. (2021, August 20). Retrieved from https://en.wikipedia.org/wiki/Complete_graph

[4] Erdős–Rényi model. (2021, August 21). Retrieved from https://en.wikipedia.org/wiki/Erdős–Rényi_model

[5] Horváth, G., Korándi, J., & Szabó, C. (2010). *Mathematics in Good Will Hunting II: Problems from the students perspective*. 1-14.

[6] Lollipop graph. (2021, May 11). Retrieved from https://en.wikipedia.org/wiki/Lollipop_graph

[7] Regular graph. (2021, August 20). Retrieved from https://en.wikipedia.org/wiki/Regular_graph

[8] Ross, S. M. (2002). *Introduction to Probability Models* (8th ed.). Amsterdam: Academic Press.

[9] Sant, G. V. (Director). (1997). *Good Will Hunting* [Film]. Be Gentlemen.

[10] Star (graph theory). (2021, October 28). Retrieved from https://en.wikipedia.org/wiki/Star_(graph_theory)

[11] Tree (graph theory). (2021, November 13). Retrieved from https://en.wikipedia.org/wiki/Tree_(graph_theory)

9 Appendix (Full Code)

```
# Imports
import networkx as nx
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from itertools import combinations, groupby
import random
```

```
def get_d(A):
    # Diagonal degree matrix of A
    D = np.diag(np.sum(A, axis=0))
    return D

def get_p(A, D):
    # Transition probability matrix from scaling all
        adjacencies with 1/degree
    P = np.matmul(np.linalg.inv(D), A)
    return P

def get_g(A):
    # Create networkx graph from adjacency matrix
    G = nx.from_numpy_matrix(A, parallel_edges=True,
        create_using=nx.MultiGraph())
    return G

def calculate_pi(P):
    # approximate stationary distribution of Markov Chain
    return [round(x, 3) for x in list(np.linalg.
        matrix_power(P, 10000)[0])]

def graph_rw_sim(nsim, n, P, G, from_pi=False):
    # nsim different random walks of length n on G
    all_walks = []
    all_cover_data = []
    ps = P.shape[0]
    if from_pi == True:
        pi = calculate_pi(P)

    for _ in range(nsim):
        walk = np.zeros(n+1)
        cover_walk = np.zeros(n+1)
        if from_pi == False:
            v0 = np.random.choice(sorted(G.nodes()), p
                =[.25, .25, .25, .25])
        else:
            v0 = np.random.choice(sorted(G.nodes()), p=pi
                )

        walk[0] = v0

        current = v0-1
        visited = [v0]
        cover_walk[0] = len(set(sorted(visited)))
```

```
        for k in range(1, n+1):
            vk = np.random.choice(range(ps), 1, p=P[
                current])
            current = vk[0]
            walk[k] = current+1
            visited.append(current+1)
            cover_walk[k] = len(set(sorted(visited)))

        all_walks.append(walk)
        all_cover_data.append(cover_walk)

    simulation_data = pd.DataFrame(all_walks).T
    cover_data = pd.DataFrame(all_cover_data).T
    return simulation_data, cover_data

def plot_graph_rw_sim(data):
    colors = ['lightcoral', 'plum', 'cornflowerblue']
    for i in range(len(data.columns)):
        plt.figure(figsize = (10, 5))
        plt.plot(data[i], color = colors[i])
        plt.title('50-Step Random Walk on Good Will
            Hunting Graph ' + str(i+1))
        plt.xlabel('discrete time-step')
        plt.ylabel('unique vertices visited')
        plt.yticks([1,2,3,4])
        filename = ('png_data/rw_sim_graph' + str(i+1) +
            '.png')
        plt.savefig(filename, dpi = 300)
        plt.show()

def graph_rw_cover_time_sim(nsim, P, G, from_pi=False):
    # nsim cover times for random walk on G
    node_set = set(sorted(G.nodes()))
    all_cover_times = []
    ps = P.shape[0]
    if from_pi == True:
        pi = calculate_pi(P)

    for _ in range(nsim):
        if from_pi == False:
            v0 = np.random.choice(sorted(G.nodes()), p
                =[.25, .25, .25, .25])
        else:
            v0 = np.random.choice(sorted(G.nodes()), p=pi
                )
```

```
            current = v0-1
            done = False
            visited = [v0]
            k = 0
            while not done:
                vk = np.random.choice(range(ps), 1, p=P[
                    current])
                current = vk[0]
                visited.append(current+1)

                if(node_set == set(sorted(visited))):
                    done = True
                else:
                    k+=1

            all_cover_times.append(k)

        return all_cover_times

    def expected_cover_time_mc(data):
        # expected cover time based on Monte Carlo
        return np.mean(data)

    def simple_update(k, pos, G, ax, rw_realization):
        # used for making a gif of a single random walk
            realization
        ax.clear()

        nx.draw_networkx(G, pos=pos, with_labels=True)
        nx.draw_networkx(G.subgraph(rw_realization[k]), pos=
            pos, node_color="red")

        ax.set_title("Time-step %d" % k)

    def gnp_random_connected_graph(n, p):
        # Generate random undirected connected graph
        edges = combinations(range(n), 2)
        G = nx.Graph()
        G.add_nodes_from(range(n))
        if p <= 0:
            return G
        if p >= 1:
            return nx.complete_graph(n, create_using=G)
        for _, node_edges in groupby(edges, key=lambda x: x
            [0]):
            node_edges = list(node_edges)
```

```
            random_edge = random.choice(node_edges)
            G.add_edge(*random_edge)
            for e in node_edges:
                if random.random() < p:
                    G.add_edge(*e)
    return G

def expected_cover_time_barplot(d):
    exp_cover_times = [x[1] for x in d.values()]
    plt.figure(figsize=(10, 5))
    plt.bar(d.keys(), exp_cover_times, color = ['pink', '
        thistle', 'powderblue', 'salmon', 'bisque', 'plum'
        , 'lightsteelblue', 'palevioletred'])
    plt.title('Barplot of Expected Cover Times')
    plt.xlabel('graph type')
    plt.ylabel('expected cover time')
    plt.savefig('png_data/cover_time_barplot.png', dpi =
        300)
    plt.show()

def cover_time_histograms(d):
    cover_time_data = [x[0] for x in d.values()]
    colors = ['pink', 'thistle', 'powderblue', 'salmon',
        'bisque', 'plum', 'lightsteelblue', 'palevioletred
        ']
    for i, key in enumerate(d.keys()):
        plt.figure(figsize=(10, 5))
        plt.hist(cover_time_data[i], bins=20, color=
            colors[i])
        plt.title("Histogram of Cover Times for " + key +
            " Graph")
        plt.xlabel("cover time bin")
        plt.ylabel("frequency")
        plt.savefig("png_data/cover_time_hist_" + key + "
            .png", dpi = 300)
        plt.show()

def main():
    # Good Will Hunting problem adjacency matrix
    A = np.array([[0, 1, 0, 1], #1
                  [1, 0, 2, 1], #2
                  [0, 2, 0, 0], #3
                  [1, 1, 0, 0]]) #4

    D = get_d(A)
    P = get_p(A, D)
```

```
G = get_g(A)

# Relabel vertices in accordance with paper
mapping = {3:1, 0:4, 1:2, 2:3}
G = nx.relabel_nodes(G, mapping)

# Simulate 3 random walks of length 50
NWALKS = 3
WALKLEN = 25
#data1 = modified_graph_rw_sim(NWALKS, WALKLEN, P, G,
    from_pi=False).astype(int)
data1, data2 = graph_rw_sim(NWALKS, WALKLEN, P, G,
    from_pi=False)
plot_graph_rw_sim(data2.astype(int))
# plot_graph_rw_sim(data2)

# Animate and save the random walk realizations
pos = nx.spring_layout(G)
figure, axes = plt.subplots(figsize=(10, 10))

for i, col in enumerate(data2.columns):
    ani = animation.FuncAnimation(figure,
        simple_update, frames=WALKLEN+1, fargs=(pos, G
        , axes, (data1.astype(int))[col]))
    ani.save("./rw_animation_data/rw_realization" +
        str(i+1) + ".gif", writer="Pillow", dpi = 300)

# Cover time simulations
# First use Good Will Hunting graph with random
    uniform start and random stationary start
# Then more cover time simulations with other graphs
    of size 4 and different structures to analyze
    relative cover times
all_cover_time_data = {}
NSIM=2500

G_complete = nx.complete_graph(4)
G_lollipop = nx.lollipop_graph(3, 1)
G_star = nx.star_graph(3)
G_tree = nx.random_tree(4)
G_regular = nx.random_regular_graph(2, 4)
G_random = gnp_random_connected_graph(4, .5)

all_graphs = {"GWH":G, "GWH_pi":G, "Complete":
    G_complete, "Lollipop":G_lollipop, "Star":G_star,
    "Tree":G_tree, "Regular":G_regular, "Random":
```

```
        G_random}
    for key in all_graphs.keys():
        if key != "GWH" and key != "GWH_pi":
            all_graphs[key] = nx.
                convert_node_labels_to_integers(all_graphs
                [key], 1)

        A_rand = np.array(nx.adjacency_matrix(all_graphs[
            key]).todense())
        D_rand = get_d(A_rand)
        P_rand = get_p(A_rand, D_rand)
        if key != "GWH_pi":
            rand_cover_time_data =
                graph_rw_cover_time_sim(NSIM, P_rand,
                all_graphs[key], from_pi=False)
        else:
            rand_cover_time_data =
                graph_rw_cover_time_sim(NSIM, P_rand,
                all_graphs[key], from_pi=True)
        rand_expected_cover_time = expected_cover_time_mc
            (rand_cover_time_data)
        all_cover_time_data[key] = [rand_cover_time_data,
            rand_expected_cover_time]

    # Plot barplot of cover times
    expected_cover_time_barplot(all_cover_time_data)

    # Plot cover time histograms for all graphs
    cover_time_histograms(all_cover_time_data)

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