Cover Times

MATH 0710 Thesis

Terry Luongo

Abstract

The cover time of a graph is the expected time it takes for a random walk to visit every vertex in a connected graph G. This thesis investigates the cover times of several classical graph structures, including complete graphs, path graphs, cycle graphs, and star graphs. Analytical techniques are employed to derive exact or approximate cover times for each type of graph, highlighting their relationships to graph order and structure. Additionally, bounds for cover times are explored, and simulations are conducted to validate theoretical results.

Contents

1 Introduction to cover times

Before we introduce the concept of cover time, we must first introduce some relevant definitions.

Definition 1.1 (Graph)**.** A graph $G(V, E)$ is a collection of vertices and edges such that each edge in the edge set E connects two vertices in the vertex set V.

Definition 1.2 (Adjacent Vertices)**.** Two vertices $u, v \in V$ are adjacent if there exists an edge $uv \in E$.

Definition 1.3 (Path)**.** A path from v_1 to v_n is a sequence of edges $v_1v_2, v_2v_3, ..., v_{n-1}v_n$ joining a set of vertices.

Definition 1.4 (Connected Graph)**.** A graph G is connected if for all vertices $u, v \in G$ there exists a path from u to v.

Definition 1.5 (Random Walk)**.** A random walk on a graph G is obtained from starting at any vertex v and then randomly traversing the graph, where at each step i all vertices $v_{j_1}, v_{j_2}, ..., v_{j_n}$ adjacent to v_i are considered with probability $\frac{1}{n}$.

Definition 1.6 (Cover Time)**.** The cover time a graph is the expected amount of steps a random walk takes to reach all vertices of a connected graph G.

We define C_i as the cover time starting from v_i and the cover time C of graph G as

$$\max_{i \in V}\{C_i\}$$

1.1 Solving for cover times via system of equations

We can solve the cover time with some careful recursion and a systems of equations. Loosely, our cases (and thus equations) are split by which vertex the walk is at, and which vertices have already been visited. This approach is only approachable by hand for very small n, and due to the combinatorial explosion cannot practically be used by computers past a certain size. Nonetheless we will demonstrate with a small example.

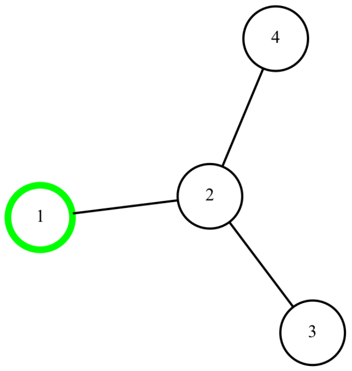


Figure 1: Cover time starting at a leaf vertex in the star graph S_4

We will demonstrate how to calculate C_1, the cover time starting from vertex 1. The approach was described in depth here Blom and Sandell's paper. [3]

1. When leaving vertex 1, we can only travel to vertex 2. So let a be the remaining time from vertex 2. We thus have
$$C_1 = 1 + a$$
2. At vertex 2, we can either travel back to 1 or to 3 or 4. If we travel back to 1, let b be the remaining time having already visited 1 and 2. If we travel to 3 or 4, by symmetry let c be the remaining time having already visited {1,2,4} or {1,2,3}. So we have
$$a = 1 + \frac{b}{3} + \frac{2c}{3}$$
3. If we are back at vertex 1, we will just go back to vertex 2 the next step with the same amount of vertices visited. So we have
$$b = 1 + a$$
4. If we are at vertex 3 or 4, we will go back to vertex 2, but critically with 3 vertices visited instead of 2. Define d as the time remaining at vertex 2 with 3 visited. Now we have
$$c = 1 + d$$

5. Finally, when at vertex 2 with 3 vertices visited, we either go to an old one with probability $\frac{2}{3}$ or visit the new one with probability $\frac{1}{3}$. So we have

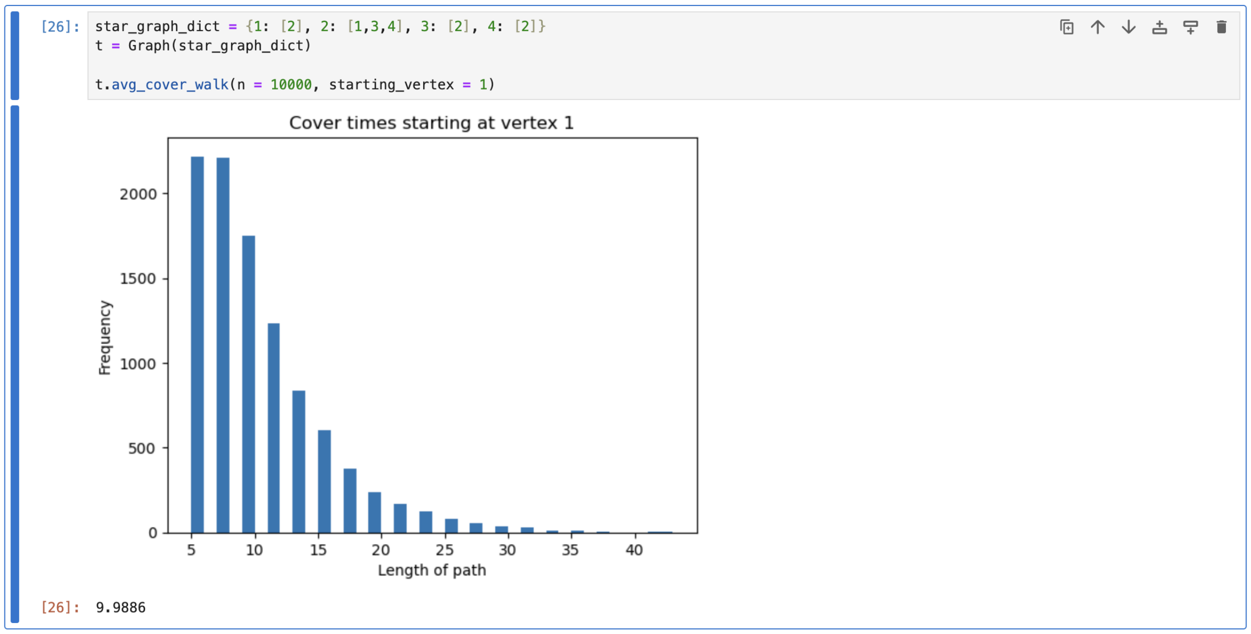
$$d = 1 + \frac{1}{3} \cdot 0 + \frac{2}{3} \cdot c$$

We thus have a system of 5 unknowns and 5 equations. Putting these into a system and solving leaves us with:

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -\frac{1}{3} & -\frac{2}{3} & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & -\frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} C_1 \\ a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The system is consistent and has a unique solution: $\{9, 8, 9, 6, 5\}$. In this case, we only care about C_1, which has value 9. Thus on average, starting at vertex 1 it will take 9 steps to reach every vertex in S_4.

This can be confirmed via simulation.



```
[26]: star_graph_dict = {1: [2], 2: [1,3,4], 3: [2], 4: [2]}
      t = Graph(star_graph_dict)

      t.avg_cover_walk(n = 10000, starting_vertex = 1)
```

```
[26]: 9.9886
```

Figure 2: Simulation of S_4 cover time starting from leaf vertex. Frequency histogram plot of cover times

Note that in this Python implementation the cover time is indexed differently. The average cover time returned here is 10, which is the average length of the random walks. A random walk of length 10 would thus correspond to 9 steps, matching with the linear algebraic solution.

Now that we have seen one method of solving for cover times, we will proceed to some classical graphs and their respective general solutions. The structure of these graphs allows for analytical solutions, and will unlock some insights that we can apply to more arbitrary graphs.

2 Cover times of P_n

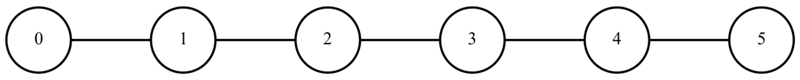


Figure 3: Finite path graph P_6 (0-indexed)

We will first look at the cover time of a path graph P_n. P_n has n vertices, where vertex i is adjacent to $i-1$ and $i+1$, for $0 < i < n$.

2.1 Similarity to linear algebraic model

We chose to examine this graph first as the method for solving it bears some resemblances to the linear algebraic method laid out above. We will have an equations for being at each vertex - so slightly different then in the linear algebraic method, where roughly each equation corresponds to a current vertex and previous vertices visited. However, the highly structured path graph allows for many cancellations, enabling a general solution on P_n.

2.2 Relation to Gambler's ruin

Calculating the cover time of P_n has some relation to a problem in classical probability: Gambler's Ruin. [8] In a simple form, the Gambler's Ruin asks for the probability a gambler starting at x dollars will reach $y > x$ or 0 (ruin) first, given a series of identical 1-dollar bets with winning probability p.[**?**] It can be shown when the winning probability is $\frac{1}{2}$ the Gambler will reach y before 0 with probability $\frac{x}{y}$. In the calculation of cover time of P_n below, we are first calculating the expected time to reach either 0 or n. This is essentially the expected time for Gambler's Ruin.

2.3 Proof

Consider the cover time of the path graph P_n from vertex i. As there is, by definition, only one path through the graph, the cover time is reached when both ends of the graph are hit. Thus to find the cover time from an arbitrary vertex, we have to reach one end of the graph, then traverse the graph to the other end.

2.3.1 Reaching 0 or n from i

Proof. First we consider reaching one end of the graph. Let S_i be a random variable indicating the amount of steps to reach vertex 0 or vertex n from i. Let $e_i = E[S_i]$. Clearly $e_0 = e_n = 0$: we already are at a desired vertex.

Consider one step in the walk, with $0 < i < n$. We will take one step and arrive with equal probability at the vertex to the left or right. The expected time can thus be represented as this step plus the expected time from the left or right:

$$e_i = 1 + \frac{1}{2}e_{i-1} + \frac{1}{2}e_{i+1}$$

Rearranging terms and doubling both sides:

$$e_{i+1} - e_i = e_i - e_{i-1} - 2$$

This is a difference equation. We can write these out for each valid i and sum to get some helpful cancellations:

$$\begin{aligned}
e_2 - e_1 &= e_1 - e_0 - 2 \\
e_3 - e_2 &= e_2 - e_1 - 2 \\
e_4 - e_3 &= e_3 - e_2 - 2 \\
\vdots\ &\ \vdots \\
e_n - e_{n-1} &= e_{n-1} - e_{n-2} - 2
\end{aligned}$$

$$e_n - e_1 = e_{n-1} - e_0 - 2(n-1)$$

Now we can add in our initial conditions: $e_n = e_0 = 0$. Additionally, by symmetry $e_1 = e_{n-1}$. So we have

$$2e_1 = 2(n-1) \implies e_1 = n-1$$

Thus, the expected number of steps to reach 0 or n from vertex 1 is $n-1$, somewhat counterintuitively. Using the difference equations and a similar method we can thus solve for e_i.

$$\begin{aligned}
e_1 - e_0 &= e_1 \\
e_2 - e_1 &= e_1 - 2 \\
e_3 - e_2 &= e_1 - 4 \\
\vdots &\quad \vdots \\
e_i - e_{i-1} &= e_1 - 2(i-1)
\end{aligned}$$

$$e_i - e_0 = ie_1 - 2\big(0 + 1 + ... + (i-1)\big)$$

$$e_i = i(n-1) - i(i-1) = i(n-i)$$

□

Thus the expected time to reach vertex 0 or n from vertex i is $i(n-i)$. This fits with the initial conditions: $e_0 = e_n = 0$. Additionally, the longest expected time is directly in the middle of the path, at $\frac{n}{2}$ (if n even). In continuous land, the parabola $x(n-x)$ reaches its maximum value of $\frac{n^2}{4}$ at $\frac{n}{2}$.

2.3.2 Reaching one end from the other

Now we have to reach the other end of the graph. Without loss of generality, we can assume we are at the 0 end of the graph: if not a relabeling fixes this.

Proof. Let T_i be the number of steps from the ith vertex to the $i+1$th.

We either take one step right with probability $\frac{1}{2}$, and are done, or go left. If we go left to the $i-1$th vertex. We must make our way back to the ith, then the $i+1$th. Thus, we have

$$T_i = 1 + \frac{1}{2} \cdot 0 + \frac{1}{2}\big[T_{i-1} + T_i\big]$$

Rearranging terms:

$$T_i = 2 + T_{i-1}$$

Additionally, we know $T_0 = 1$ (can only go right from 0th vertex). So we have

$$\begin{aligned}
T_1 &= 2 + 1 = 3 \\
T_2 &= 2 + 3 = 5 \\
\vdots &\quad \vdots \\
T_i &= 2n + 1
\end{aligned}$$

To reach vertex n from 0, we need $T_0, T_1, ..., T_{n-1}$ all to occur. Thus we have

$$\begin{gathered} E[T_0 + T_1 + ... + T_{n-2} + T_{n-1}] \\ = E[T_0] + E[T_1] + ... + E[T_{n-2}] + E[T_{n-1}] \\ = 1 + 3 + 5 + 7 + ... + n - 1 = n^2 \end{gathered}$$

$\square$

Thus the expected cover time is $i(n-1) + n^2$. We can see that the cover time at 0 and n is n^2: this makes sense as we have already reached the first end of the path.

2.4 Alternate proof for end-to-end

Proof. To calculate the time from 0 to n, we can also use our previous $i(n-i)$ result on P_{2n}. First, we establish a one-to-one correspondence between walks from 0 to n on P_n and walks from n to 0 or $2n$ on P_{2n}. We can simply consider distance from n on P_{2n}. Traveling n away leaves us at 0 or n, both of which are equivalent to going from 0 to n on P_n. Thus, $i(n-1) = n(2n-n) = n^2$. $\square$

2.5 Extending finite results to partial covers on an infinite graph

We can generalize our finite solution on P_n to a partial cover of the number line: intuitively how long does it take to reach n vertices on the number line during a random walk?

Proof. Let T_k be the steps to visit k points on the number line (irrespective of starting vertex). We can write T_k as $Y_1 + Y_2 + ... + Y_{k-1}$, where Y_i is the time to visit the ith point after already having $i-1$. Note that $Y_1 = 1$ (we have neither visited neighbor).

Denote $h_i = E[Y_i]$. Thus, $E[T_k] = h_1 + h_2 + ... + h_{n-1}$.

Note that when we just reach the *ith* point, we must be at the left or rightmost vertex in the set of points visited. Without loss of generality, assume we are at the leftmost. To reach one more point we must either go immediately left, or go right i vertices more. This is analogous to the linear case of $n = i + 1$ starting at vertex 1.

We can thus plug in our formula for the cover time: $i(n-i)$, we get $1 \cdot (i + 1 - 1) = i$: it's expected to take i steps to reach $i+1$ vertices after getting i vertices.

$$E[T_k] = h_1 + h_2 + ... + h_{n-1} = 1 + 2 + ... + (n-1) = \frac{(n-1)n}{2}$$

$\square$

To reach n vertices in total, the partial cover time of the infinite line is $\frac{(n-1)n}{2}$.

2.6 Reduction of C_n to P_n

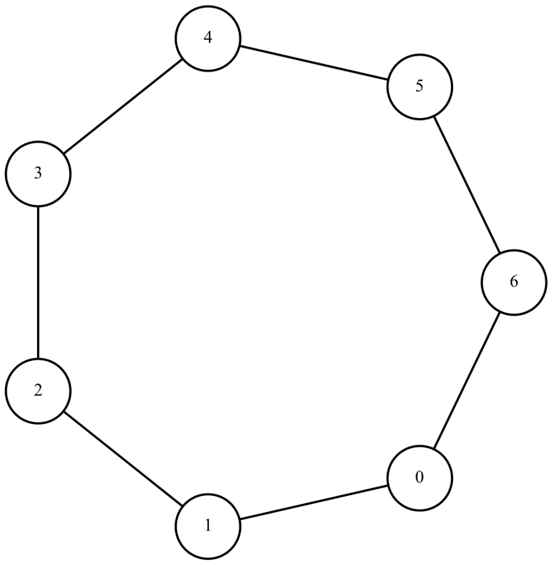


Figure 4: Cyclic/Cycle graph C_7

Proof. We can establish a correspondence between a cover walk on C_n and a partial walk to n vertices on the number line. We can label each vertex with multiple labels: $\{i, i+n, i+2n, ..., i+kn \forall k \in \mathbb{Z}\}$. In doing this we establish a mapping between the points on the number line and the vertices while preserving the cyclic nature: vertex 6 is connected to vertex 0 because vertex 0 can also be labelled as 7. In this way, covering C_n is equivalent to covering n vertices on the number line: the expected time is $\frac{(n-1)n}{2}$. [2] $\square$

It's interesting to compare the differences between cover time on C_n and P_n. The addition of the edge between the first and last vertex reduces the cover time by at least a factor of 2: the path cover time is minimum n^2 when starting at an edge. This makes sense as the diameter of C_n is half of that of P_n, and there exist bounds on the cover time based on the diameter of a graph. (need a citation here and a definition of diameter).

We will next turn our attention to another highly structured graph: K_n. Once again, the structure will significantly aid in analysis and a generalized solution. Through comparing cover times of P_n (sparsely connected) and K_n (by definition maximally connected) we can see the impact of degree on cover times.

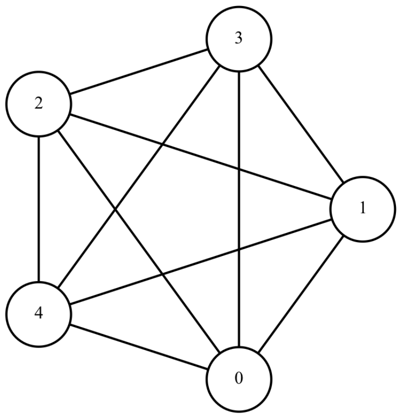


Figure 5: Complete graph K_5

3 Cover times of K_n

We will next look at the cover time of the complete graph K_n. K_n has n vertices, where vertex i is adjacent to all $(n-1)$ others, for a total of $\frac{1}{2}n(n-1)$ edges.

3.1 Relation to coupon collector and drawing from a urn of balls

Cover time on a random walk on K_n is highly analogous to problems in classical probability. As you can get to any other vertex at each step in the walk, this is akin to drawing balls from an urn with replacement. When we have sampled all of the balls, we have visited all of the vertices. A common formulation of this problem is the coupon collector's problem.

Problem 3.1. *Coupon Collector's Problem: If we have n coupons in total, how many boxes of cereal do we have to open before we have obtained a complete set? [7]*

There is one key difference: K_n does not contain any loops - we cannot get to any vertex from itself. However, we can draw the same coupon or ball twice in a row.

3.2 Proof

We can use the same technique as in the path graph - breaking up the cover time into sequential times to visit the next unvisited vertex.

Proof. Let T be the cover time of K_n. We have $T = t_1 + t_2 + ... + t_{n-1}$, where t_i is the time to visit $i+1$ vertices having already visited i. Each component t_i is independent as they are without respect to which vertices have been visited.

To find t_i, note that from any vertex in K_n there exists an edge to $n-1$ other vertices. If we have visited i vertices already, there will be new $(n-1)-(i-1)$ vertices we can choose (we have already visited our current vertex). Note this is a geometric distribution - we are counting failures until success with probability $\frac{(n-1)-(i-1)}{n-1}$. So we have $t_i \sim \text{Geom}\left(\frac{n-i}{n-1}\right)$. The expected value of a geometric distribution is $\frac{1}{p}$, so we have $E[t_i] = \frac{n-1}{n-i}$. Thus

$$\begin{aligned} E[T] &= E[t_1 + t_2 + ... + t_{n-1}] = E[t_1] + E[t_2] + ... + E[t_{n-1}] \\ &= \frac{n-1}{n-1} + \frac{n-1}{n-2} + \frac{n-1}{2} + ... + \frac{n-1}{1} \\ &= (n-1)\left(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n-1}\right) = (n-1)H_{n-1} \end{aligned}$$

□

Note that although this problem bears a lot of similarity to the coupon collector, its cover time, nH_n is higher - in a random walk on K_n we cannot visit the same vertex twice in succession, increasing the probability of a helpful step. As a Vermonter would say: “you can’t get there from there”.

3.3 Mapping coupon collector result onto cover time of K_n

We can derive the $(n-1)H_{n-1}$ cover time on K_n result from the general coupon collector’s solution by accounting for the loops and starting vertex. Initially we have

$$E[T] = nH_n = n(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$

First, note that we need to collect the first coupon (takes 1 turn), but we start with one vertex visited. This allows us to remove the last term from the harmonic number (we flipped the order).

Second, we need to account for successive visits to the same coupon. Note that at each turn the probability of a successive visit is $\frac{1}{n}$. Thus the number of nonsuccessive visits when visiting the ith vertex follows a geometric distribution with probability $\frac{n-1}{n}$. Thus the expected value is $\frac{n}{n-1}$. We condition the coupon collector result with this - that we do not have any successive visits. We thus divide each term in the sum by $\frac{n}{n-1}$:

$$\left(\frac{n-1}{n}\right)n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n-1} + \cancel{\frac{1}{n}}\right) = (n-1)H_{n-1}$$

3.4 Partial covering times and rates as $n \to \infty$: approximations and convergence

One might be interested in the rate we cover new vertices, especially as $n \to \infty$. We can turn to classical occupancy problems for this. The amount of vertices on K_n empty after r steps is analogous to placing r balls randomly into n boxes, counting the number of empty boxes. Here we will look at case where $r = n$ - the amount of unvisited vertices on K_n after n steps in our random walk. [4]

We want an analogue to partial cover time - however instead of calculating the expected time to visit a specified number of vertices, we are calculating the expected number of vertices visited after a specified length walk.

To find asymptotic behavior, we are fixing $\frac{r}{n} = c$, taking $n \to \infty$. The expected number of times we have have visited each vertex turns into a Poisson distribution with mean c. Thus, by symmetry the expected number of vertices not visited $= n \cdot P(0)$. This asymptotic behavior occurs when $ne^{-\frac{r}{n}} \to \lambda$. Before we examine the limiting case, when $n \to \infty$, we need general equations representing the probability of vertices being unvisited.

To start, note that for specified vertices in an index set $I = \{i_1, i_2, ..., i_k\}$, the probability of those specific vertices being unvisited is $\left(1 - \frac{k}{n}\right)^r$ after r steps. We can use this result to calculate the probability all vertices were reached by using the Inclusion-Exclusion Principle probabilistically:

$$\mathbb{P}\big(\bigcup_{i=1}^{n} A_i\big) = \sum_{k=1}^{n} \Bigg((-1)^{k-1} \sum_{\substack{I \subseteq \{1,\ldots,n\} \\ |I|=k}} \mathbb{P}\big(A_I\big)\Bigg)$$

A_i in this case represents vertex i being unvisited.

Here we have $\mathbb{P}\big(\bigcup_{i=1}^{n} A_i\big) = p_0(r, n)$ - the probability that we have 0 unvisited vertices after r steps on K_n. Applying P.I.E along with our formula for any index set I being all unvisited:

$$p_0(r_n) = \sum_{k=1}^{n} (-1)^{k-1} \binom{n}{k} (1 - \frac{k}{n})^r$$

We can use this probability to get a more general expression: the probability that exactly m vertices are unvisited.

$$p_m(r, n) = \binom{n}{m} (1 - \frac{m}{n})^r p_o(r, n - m)$$

where the three terms respectively indicate

1. All possible assignments of m unvisited vertices
2. The probability that index set is unvisited
3. The probability that none of the remaining $n-m$ vertices are unvisited

We are interested in the case where $n \to \infty$: we want an expression for $p_m(r,n)$ as $ne^{-\frac{r}{n}} \to \lambda$

To examine the asymptotic behavior, we will have to show that $\binom{n}{m}(1-\frac{m}{n})^r \to \frac{\lambda^m}{m!}$. We will do this by showing both inequalities hold.

Proof. The first step is showing that $\binom{n}{m}(1-\frac{m}{n})^r \leq \frac{\lambda^m}{m!}$. First note that $(1-x) \leq e^{-x}\,\forall\, x$. Additionally, we have the following lemma:

Lemma 3.2. $\binom{n}{m} \leq \frac{n^m}{m!}$

Proof.
$$\frac{n!}{m!(n-m)!} = \frac{1}{m!}n(n-1)...(n-m) \leq \frac{1}{m!}n \cdot n... \cdot n = \frac{n^m}{m!}$$
□

So we have
$$\binom{n}{m}(1-\frac{m}{n})^r \leq \frac{n^m}{m!}e^{-\frac{rm}{n}} = \frac{(ne^{-\frac{r}{n}})^m}{m!} = \frac{\lambda^m}{m!}$$

Now we will show the $\geq$ side. First a similar lemma.

Lemma 3.3. $\binom{n}{m} \geq \frac{(n-m)^m}{m!}$

Proof. $\binom{n}{m} = \frac{n!}{m!(n-m)!} = \frac{1}{m!}\big(n(n-1)...(n-m)\big) \geq \frac{1}{m!}\big((n-m)(n-m)...(n-m)\big) = \frac{(n-m)^m}{m!}$ □

So we have
$$\binom{n}{m}\big(1-\frac{m}{n}\big)^r \geq \frac{(n-m)^m}{m!}\big(1-\frac{m}{n}\big)^r$$
$$= \frac{1}{m!}\big(1-\frac{m}{n}\big)^r\big(n(1-\frac{m}{n})\big)^m = \frac{n^m}{m!}\big(1-\frac{m}{n}\big)^{r+m}$$

Note that $(1-\frac{m}{n})^m \to 1$ as $n \to \infty$, and $\frac{1}{m!}$ is constant. So we are only concerned with $n^m(1-\frac{m}{n})^r$.

We will take the logarithm of this expression:

$$\log(n^m(1-\frac{m}{n})^r) = m\log(n) + r\log(1-\frac{m}{n})$$

Examining the Taylor series of $\log(t)$ for $0 \leq t \leq \frac{1}{2}$, we have

$$\log(1-t) = -t - \frac{t^2}{2} - \frac{t^3}{3}.... = -t - \frac{t^2}{2}\big(t + \frac{2}{3}t + \frac{2}{4}t^2 + ...\big)$$

$$\geq -t - \frac{t^2}{2}\big(1 + t + t^2 + t^3 + ...\big) \geq -t - \frac{t^2}{2}\big(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...\big)$$

$$= -t - t^2$$

Note that $0 \leq t \leq \frac{1}{2}$ as $\frac{m}{n} \leq \frac{1}{2}$: $n \to \infty$ and m is fixed. So we have

$$m\log n + r\log(1-\frac{m}{n}) \geq m\log n - r\frac{m}{n} - r(\frac{m}{n})^2$$

.

Now, note that $ne^{-\frac{r}{n}} \to \lambda$. We will take the logarithm of both sides and solve for r:

$$\log(ne^{-\frac{r}{n}}) \to \log(\lambda)$$

$$\log(n) - \frac{r}{n} \to \log(\lambda)$$

$$\log(n) - \log(\lambda) - \frac{r}{n} \to 0$$

$$\frac{1}{n}(n\log(n) - n\log(\lambda) - r) \to 0$$

So we have $n\log(n) - n\log(\lambda) - r = o(n)$ by definition. Thus, $r = n\log(n) - n\log(\lambda) - o(n)$. Plugging this in for r into $r(\frac{m}{n})^2$ leaves us with

$$r(\frac{m}{n})^2 = \frac{m^2\log(n)}{n} - \frac{m^2\log(\lambda)}{n} - \frac{o(n)}{n}$$

This goes to 0 as $n \to \infty$ as $\frac{o(n)}{n} \to 0$, $\frac{\log(n)}{n} \to 0$, and $\frac{1}{n} \to 0$.

We can multiply the expression for r by $\frac{m}{n}$ to get $\frac{rm}{n} = m\log(n) - m\log(\lambda) - \frac{o(n)m}{n}$. Rearranging this leaves us with

$$m\log(n) - r\frac{m}{n} = m\log(\lambda) - \frac{o(n)m}{n}$$

$$\implies m\log(n) - r\frac{m}{n} \to m\log(\lambda) = \log(\lambda^m) \text{ as } n \to \infty$$

Thus we have limiting bounds in both directions: there is convergence.

$$\liminf_{n\to\infty} \binom{m}{n} \left(1 - \frac{m}{n}\right)^r \geq \lambda^m$$

Finally, by invoking the Dominating Convergence Theorem (DCT), as $ne^{-\frac{r}{n}} \to \lambda$ we get

$$p_0(r, n) = \sum_{k=0}^{\infty} (-1)^k \frac{\lambda^k}{k!} = e^{-\lambda}$$

□

4 Cover times of $S_{r,v}$

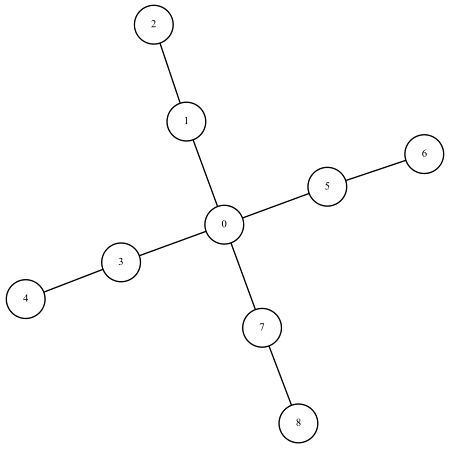


Figure 6: Star graph $S_{4,2}$ - 4 rays of 2 vertices

4.1 Proof

Finally, we will examine the cover time of the generalized star graph. $S_{r,n}$ has r rays of v vertices each emanating from a central vertex. As traversing the simple star graph $S_{r,1}$ resembles traversing the complete graph (only that we must go back to the central vertex each time, doubling our steps), the star graph is effectively a combination of the path graph and the complete graph.

Proof. Let T be the time to cover star $S_{r,v}$. We will break up the cover time into independent variables

$$T = F_0 + R_1 + F_1 + R_2 + ... + R_{r-1} + F_{r-1}$$

where F_i is the time to reach the $i + 1$th end of a ray having already seen the end of i distinct ends of rays, and R_i is the time to return to the center after being at the ith end of ray. [5]

Note that $R_i = R_j \; \forall \, i, j$: the time to return to center from a ray's end is independent of ray. As each ray has v vertices, this is v^2 as proven earlier.

F_i separates into two cases:

1. We go directly to a ray's end we haven't seen yet. Denote W_i as the time to get from the center to that ray's end

2. We go to another ray's end first that we have already seen. In this case, we get to that end, return to center, and are in same position we were in to start. Denote S_i and R_i respectively as the time to get to the previously seen ray's end and return.

For F_i, the time to visit $i+1$th ray's end having seen i, we have a $\frac{i}{r}$ chance of visiting an old ray first, and a $\frac{r-i}{r}$ chance of a new one first. Thus we combining the two cases with this probability and our random variables we have introduced:

$$F_i \overset{d}{=} \frac{i}{r}\Big(S_i + R_i + F_i'\Big) + \frac{r-i}{r}\Big(W_i\Big)$$

Note that S_i, R_i, W_i are all random variables representing the time to traverse an n-path. Thus, they will all have the same expected value, v^2. Additionally, $F_i \overset{d}{=} F_i'$ as visiting a new ray's end is independent of how many ray's end we superfluously visit - the expectation will thus be equivalent. Now we can calculate the expected value of F_i:

$$\begin{aligned} rE[F_i] &= i(2v^2 + E[F_i]) + (r-i)v^2 \\ (r-i)E[F_i] &= v^2(2i + r - i) \\ E[F_i] &= v^2\frac{r+i}{r-i} \end{aligned}$$

So, we now have

$$T_s = F_0 + R_1 + ... + R_{r-1} + F_{r-1} = (r-1)R_i + \sum_{i=0}^{r-1} v^2\frac{r+i}{r-i}$$

$$\begin{aligned} T_s &= v^2\Big(r - 1 + \sum_{i=1}^{r-1}\frac{r+i}{r-i}\Big) = v^2\Big(\sum_{i=0}^{r-1}\big(1 + \frac{r+i}{r-i}\big) - 1\Big) \\ &= v^2\Big(\sum_{i=0}^{r-1}\frac{r-i+r+i}{r-1} - 1\Big) = v^2\Big(\sum_{i=0}^{r-1}\frac{2r}{r-i} - 1\Big) \\ &= v^2\big(2r\sum_{i=0}^{r-1}\frac{1}{r-i} - 1\big) \\ T_s &= v^2\big(2r\sum_{i=1}^{r}\frac{1}{i} - 1\big) = 2v^2 rH_r - v^2 \end{aligned}$$

□

As the star graph is essentially a combination of path graphs and a completely-connected vertex, we can see the cover time is the product of both components' cover times. We need $2v^2$ to traverse each ray to the end and back, rH_r to get each ray, and we subtract v^2 as we don't need to go back to the center once we have gotten to the final ray's end .

5 Bounds on Cover Time

Now that we have seen the cover time for multiple types of graphs, we can begin to analyze the order of cover time with respect to the number of vertices. First we will comapre the orders for our previously established results, then we will look at worst-case scenarios. What is the slowest cover time we can get on n vertices?

5.1 Order of Cover Time for Classical Graphs

Type of Graph	**Cover Time**	**Order**
Path Graph P_n	$i(n-1)+n^2$	n^2
Cycle Graph C_n	$\frac{n(n-1)}{2}$	$\frac{1}{2}n^2$
Star Graph $S_{r,v}$	$v^2(2rH_r - 1)$	$2v^2 r\log(r) \approx 2n\sqrt{n}\log(\sqrt{n})$
Complete Graph K_n	$(n-1)H_{n-1}$	$n\log(n)$

Table 1: Cover times and orders for various types of graphs.

As you can see in the above chart, the order of cover times slowest to fastest goes from path, cycle, star, to complete. Adding additional edges tends to speed up the cover time - the path graph's minimum edges for a connected graph on n vertices gets us order n^2 compared to the maximum edges on a complete graph's $n\log n$. Additionally, some key edge additions can substantially decrease cover time - connecting the start and end on a path to form a cycle cuts the cover time in half. Comparing these results may lead the reader to ask what the worst case cover time is? Are there cover times on the order of n^3, even n^4? [2]

5.2 Worst Case Cover Time - The Lollipop Graph

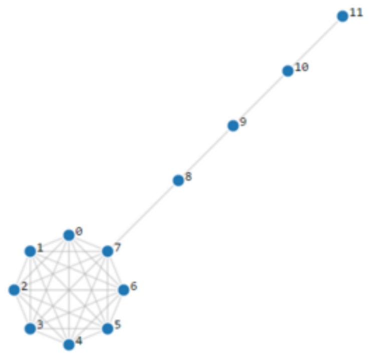


Figure 7: Lollipop graph $(\frac{2n}{3}, \frac{n}{3})$ - terrible for cover time[3]

It turns out the lollipop graph is just about the worst arrangement of n vertices with respect to cover time: on the order of $\frac{4}{27}n^3$ [6] The lollipop graph consists of $\frac{2n}{3}$ vertices arranged in a complete sub-graph, connected by a bridge to $\frac{n}{3}$ vertices in a path.

To demonstrate that the lollipop graph is as bad as it gets, we need to look at bounds on cover time.

5.3 Initial Bounds

Having a bound on a graph in terms of number of vertices and edges is desirable: we can quickly glean the worst-case behavior. In 1979, Aleliunas[1] showed $C \leqslant 2e(n-1)$. As the maximum number of edges in n-graph is $\frac{1}{2}n^2$, this bounds cover time to order n^3. The Lollipop graph with this bound gives us $\frac{4}{9}(n-1)^2(n+2)$: this is close to the $n^2(n-1)$ absolute bound on n-graph.

More recent research tends to focus on cover time of randomly generated graphs for large n.

5.4 Bounds on Diameter

Definition 5.1 (Diameter)**.** The diameter of a graph, D is the maximum distance between any two vertices u, v on a graph G.

Out of all the literature I reviewed, diameter was the one graph property that was useful for bounding the cover time of a graph. Zuckerman[9] in 1989 proved $C \leqslant De \ln n$.

This makes sense - it will take a while to randomly traverse between two points if they are very far away. A graph like K_n of diameter 1 enables very quick travel - we can get anywhere from anywhere in one step.

6 Chess

We can apply a cover time analysis to the chessboard. Given a specific chess-piece, eg. a knight or a king, how long will it take to cover the whole board during a random walk? We will connect this to bounds on cover time based on diameter, as well as vertex degree.

6.1 What is the cover time of a knight? Does starting vertex matter?

Using the Python cover time simulator I developed, I created adjacency matrices and later graphs for an 8x8 chessboard under four different movement patterns: a king, queen, rook, and knight. Only the rook and knight's graphs make any sense visually - the knight's is displayed here for reference. The rook and queen are very jumbled as "adjacent" vertices for a queen don't quite correspond to "adjacent" squares - the queen can get across the entire board in one move.

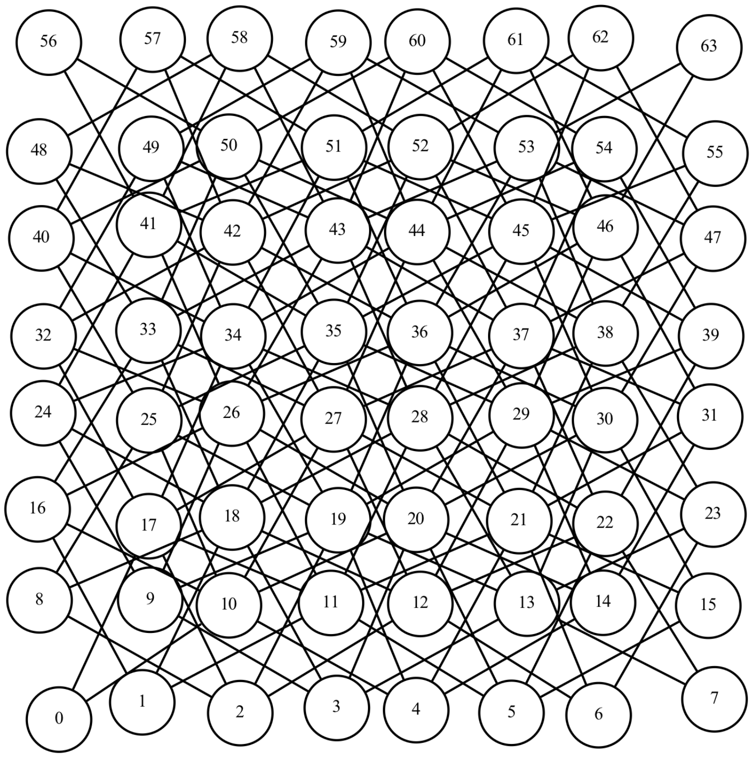
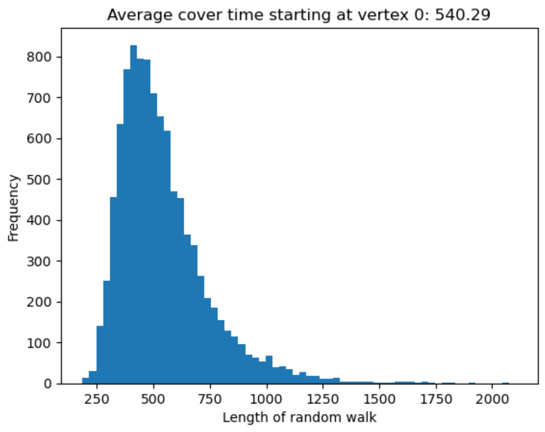
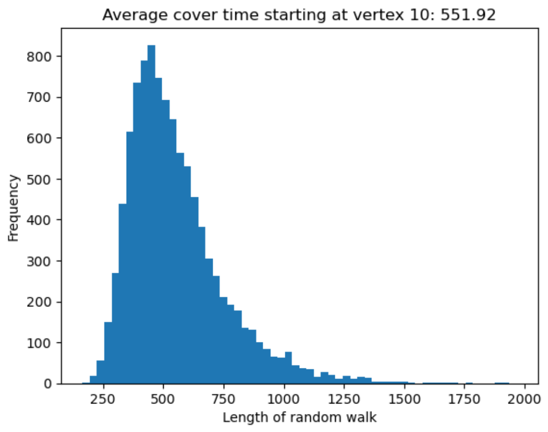


Figure 8: A knight's adjacency graph on 8x8 chessboard

6.2 Cover time of the knight - is it quicker to start in the corner?



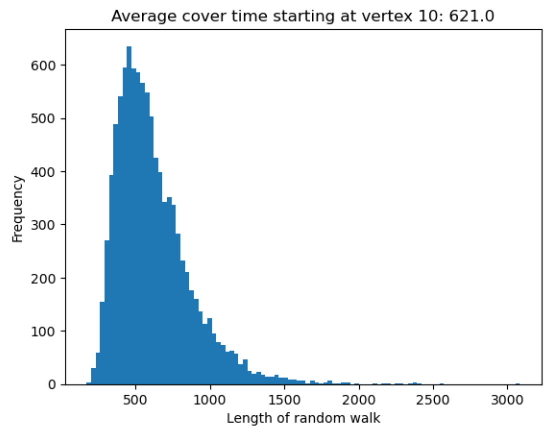


As we can see, it takes the knight around 540 moves to cover the board starting at a corner, and 552 starting at vertex 10 (I chose vertex 10 as it is adjacent to the corner 0). This gives us a ratio of about $\frac{550}{64} \approx 8.5$ - each square is visited around 8.5 times. We can see the distribution is right-skewed - a random walk rarely takes significantly quicker than average to cover all vertices, but frequently takes much longer.

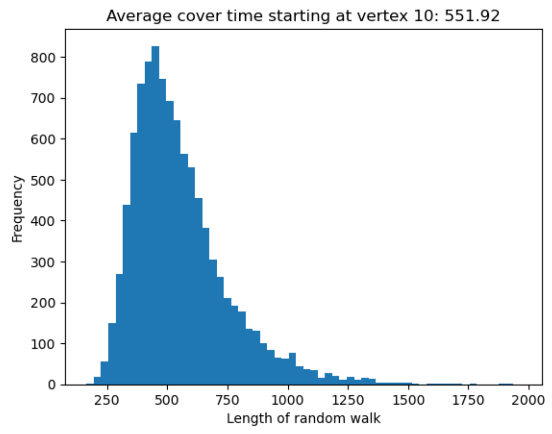
As the corners of the board have low degree (only 2 moves in and out), a random walk visits there with relatively low probability. Thus starting at a corner decreases the moves required to cover the board by about 12 on average. Additionally, the right tail is longer when we don't start in a corner - there are particularly more walks of length longer than 1250 in the second graph. This could imply the hard-to-reach vertices with lower degree are usually reached later in the walk.

Now we will compare the cover time of the knight to the other chess pieces.

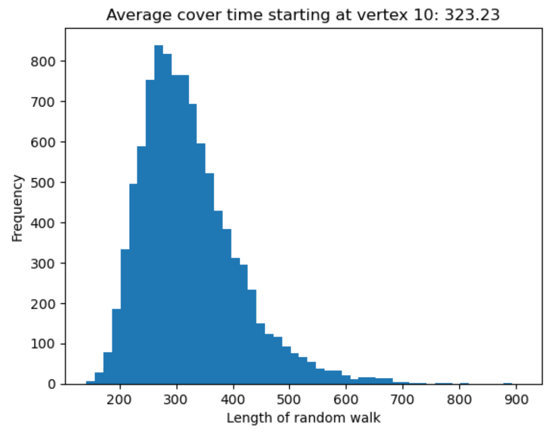
6.3 Cover times of different chess pieces



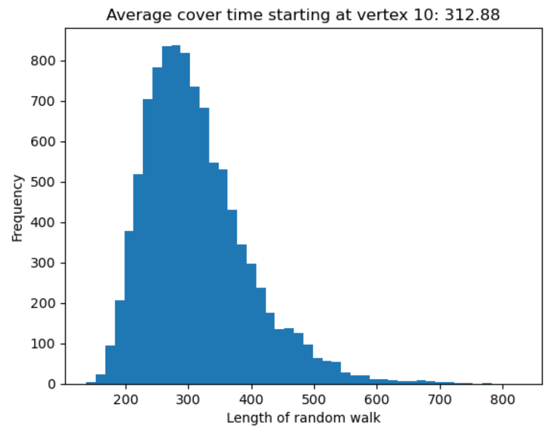
(a) King



(b) Knight



(c) Rook



(d) Queen

Figure 9: Cover times of different chess pieces (King, Knight, Rook, and Queen) on a chessboard over 10,000 iterations.

As indicated in the figure, there is a substantial difference in the cover time between the king and knight and the cover time of the rook and queen. The ability of the latter pieces to cross the board in one move reduces the cover time by around 40%. The diameter of the chess board - the most moves to traverse between any two squares - under the king is 8, compared to 2 for the rook and queen. The queen and rook's cover times of 312 and 323 respectively approach the diameter's lower bounding result of K_{64}: $nH_n \approx 303$.

7 Concluding Remarks

We have now seen cover time derivations for four types of classical graphs: path, cycle, complete, and a generalized star graph. Furthermore, we have examined bounds on cover time and worst-case behavior. Finally, we looked at an “application” of cover time through examining how different chess pieces would cover the chess board.

Cover time is certainly not the most studied topic in graph theory: to my knowledge only Alan Frieze of Carnegie Mellon U. and Colin Cooper of U. London are currently prioritizing this in their research. The applications are hard to find as well: perhaps one could imagine an infectious disease that only one person can have at a time - perhaps a game of tag. That being said, its intuition and explainability make it very interesting to consider. I certainly enjoyed my semester studying it.

8 Acknowledgments

First and foremost, I would like to thank my thesis advisor Professor Peterson for his help and support over the course of this past semester. His guidance on selecting sources and then understanding and interpreting them proved invaluable for much of this project.

I would also like to thank my peers in MATH 0710 - it was great to hear everyone’s progress and see their work come to fruition. Finally, I would like to thank the various friends and family members for their never-ending support.

References

[1] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. *21st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 218–223, 1979.

[2] Gunnar Blom, Lars Holst, and Dennis Sandell. *Problems and Snapshots from the World of Probability.* Springer, 1994. SpringerLink.

[3] Gunnar Blom and Dennis Sandell. Cover times for random walks on graphs. University of Oregon, Department of Mathematics. Accessed: 2024-11-04.

[4] Rick Durrett. *Probability: Theory and Examples.* Cambridge University Press, 2019. 5th Edition.

[5] Michael H. Duyzend, Rebecca L. Ferrell, and Miranda J. Fix. Cover times of random walks on finite graphs. Carleton College, Department of Mathematics, 2008. Accessed: 2024-12-01.

[6] Daniel Wallén. Cover times of random walks on graphs. Uppsala University, Diva Portal, April 2010. Accessed: 2024-11-04.

[7] Wikipedia contributors. Coupon collector's problem — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Coupon_collector%27s_problem`, 2024. [Online; accessed 9-December-2024].

[8] Wikipedia contributors. Gambler's ruin — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Gambler%27s_ruin`, 2024. [Online; accessed 9-December-2024].

[9] David Zuckerman. Covering times of random walks on bounded degree trees and other graphs. *Journal of Theoretical Probability*, 2(2):147–158, 1988.

A Code Appendix

A.1 Custom Graph Class

```python

class Graph(object):

    def __init__(self, graph_dict, graph_type = "user"):
        self.dict = graph_dict
        self.n = len(graph_dict.keys())
        self.graph_type = graph_type
        self.engine = self.map_engine()
        self.graphviz = self.make_graphviz()


    @classmethod
    def make_classic(cls, n, graph_type):
        graph_dict = {}

        if graph_type == "complete":
            graph_dict = make_complete_graph(n)
        elif graph_type == "cycle":
            graph_dict = make_cycle_graph(n)
        elif graph_type == "linear":
            graph_dict = make_linear_graph(n)

        return cls(graph_dict, graph_type)


    def map_engine(self):
        engine_dict = {"complete": "circo", "cycle": "circo", "linear": "
dot", "user": "neato"}
        return engine_dict[self.graph_type] if self.graph_type in
engine_dict.keys() else "neato"



    def make_graphviz(self):

        g = graphviz.Graph(engine = self.engine) #
        keys = list(self.dict.keys())
        visited = []

        g.attr('node', width='0.4', height='0.4', fixedsize='true',
fontsize='8')

        if self.graph_type == "linear":
            g.attr(rankdir='LR')  # Left to Right layout

        for i in range(len(keys)):
            a = keys[i]
            for j in range(len(self.dict[a])):
                if (self.dict[a][j], a) not in visited:
                    g.edge(str(a), str(self.dict[a][j]))
                    visited.append((a, self.dict[a][j]))
        return g

    def make_path_gif(self, gif_length = None, path = None):
        if path == None:
            path = g.cover_walk()

        if gif_length == None:
            gif_length = self.n

        filepaths = []
        visited = set()

        for i in range(len(path)):
            svg_path = 'temp/' + str(i)
            filepaths.append(svg_path + '.svg')

            #current point
            if path[i] not in visited:
                self.graphviz.node(str(path[i]), color = 'green', penwidth
    = '3')
                self.graphviz.render(svg_path, format='svg', cleanup=True)

                visited.add(path[i])

            else:
                self.graphviz.node(str(path[i]), color = 'green', style = '
    filled', fillcolor='gray', penwidth = '3')
                self.graphviz.render(svg_path, format='svg', cleanup=True)

            self.graphviz.node(str(path[i]), style = 'filled', fillcolor =
    'gray', color = 'black', penwidth = '1')

        for vertex in visited:
            self.graphviz.node(str(vertex), color = 'black', fillcolor = '
    transparent', penwdith = '1')

        create_gif(filepaths, "output.gif", gif_length)

        shutil.rmtree("temp")


    def display(self):
        svg_path = 'temp'
        self.graphviz.render(svg_path, format='svg', cleanup=True)
        display(SVG(svg_path + ".svg"))

    def export_png(self, filename = "export"):
        self.graphviz.attr(dpi='300')
        self.graphviz.render(filename, format='png', cleanup=True)

    def cover_walk(self, starting_vertex = None, print_out = True):
        if (starting_vertex == None):
            starting_vertex = list(self.dict.keys())[0]

        if (starting_vertex < 0 or starting_vertex > self.n):
            raise ValueError("The starting vertex cannot be negative or higher than n.")

        path = [starting_vertex]

        visited = set()
        visited.add(starting_vertex)

        cur = starting_vertex

        while len(visited) != self.n:
            next_vertex = random.choice(self.dict[cur])
            path.append(next_vertex)
            visited.add(next_vertex)

            cur = next_vertex

        if print_out:
            print("Cover Time: " + str(len(path)))
            print(path)

        return path


    def avg_cover_walk(self, n, starting_vertex = None, bin_size = None, filename = ""):
        # make histogram and display
        lengths = [len(self.cover_walk(starting_vertex, False)) for _ in range(n)]

        if bin_size == None:
            plt.hist(lengths)
        else:
            plt.hist(lengths, bins = range(min(lengths), max(lengths) + 1, bin_size))

        plt.title("Average cover time starting at vertex " + str(starting_vertex) + ": " + str(round(sum(lengths) / n, 2)))
        plt.xlabel("Length of random walk")
        plt.ylabel("Frequency")
```

```python
        plt.savefig(filename + self.graph_type + str(self.n) + "_" + str(n)
 + "iter" + ".png", bbox_inches = "tight")
        plt.show()

        return sum(lengths) / n
```

Listing 1: Custom Graph Class in Python for visualization and simulation - visualization using python wrapper for graphviz

```python
def make_complete_graph(n):
    graph_dict = {}
    for i in range(n):
        graph_dict[i] = []
        for j in range(n-1):
            graph_dict[i].append((i + j + 1) % n)
    return graph_dict


def make_cycle_graph(n):
    graph_dict = {}
    for i in range(n):
        graph_dict[i] = []
        graph_dict[i].append((i + 1) % n)
        graph_dict[i].append((i - 1) % n)
    return graph_dict

def make_linear_graph(n):
    graph_dict = {}
    for i in range(n-1):
        graph_dict[i] = []
        graph_dict[i].append(i + 1)
    graph_dict[n-1] = []
    for i in range(n-1):
        graph_dict[i+1].append(i)
    return graph_dict
```

Listing 2: Functions for generating adjacency lists for different types of graphs

```python
def is_valid_move(x, y, n):
    """Check if the move is within bounds"""
    return 0 <= x < n and 0 <= y < n

# adapted from https://medium.com/@davidlfliang/intro-python-algorithms-
  knights-tour-problem-ab0a27a5728c
# got a little help with GPT to adapt to king, rook, queen eficiently
def make_chessboard_dict(n, chesspiece = "knight"):

    assert chesspiece in {"knight", "king", "rook", "queen"}, "Valid
  Chesspieces are Knight, King, Rook, Queen"

```

```
    moves = {
        "knight": [
            (2, 1), (1, 2), (-1, 2), (-2, 1),
            (-2, -1), (-1, -2), (1, -2), (2, -1)
        ],
        "king": [
            (1, 0), (0, 1), (-1, 0), (0, -1),
            (1, 1), (-1, 1), (-1, -1), (1, -1)
        ],
        "rook": [
            (1, 0), (0, 1), (-1, 0), (0, -1)
        ],
        "queen": [
            (1, 0), (0, 1), (-1, 0), (0, -1),
            (1, 1), (-1, 1), (-1, -1), (1, -1)
        ]
    }

    graph_dict = {}
    piece_moves = moves[chesspiece]

    for i in range(n * n):
        x = i % n
        y = i // n

        graph_dict[i] = []

        for dx, dy in piece_moves:
            step = 1
            while True:
                new_x, new_y = x + dx * step, y + dy * step
                if is_valid_move(new_x, new_y, n):
                    graph_dict[i].append(new_y * n + new_x)
                    if chesspiece in {"king", "knight"}:  # these do not
 repeat moves
                        break
                    step += 1
                else:
                    break

    return graph_dict
```

Listing 3: Functions for chess example

B Github Link

My full code can be found at https://github.com/terryluongo/cover_time.