Computational Graph Algorithms for Markov Chains

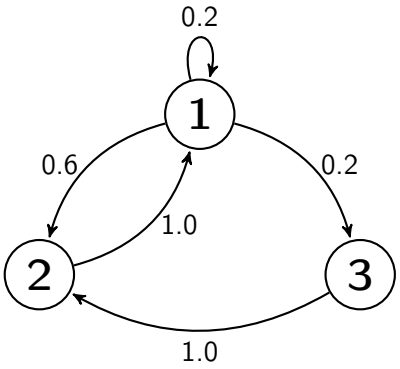
MTH 4125. Raasikh Shahid

1. Introduction

A sequence of random variables $X_1, X_2, \ldots$ defined on a common probability space Ω is called a discrete-time Markov chain if it holds the following property.

$$P[X_{n+1} = j \mid X_n = i] = P[X_{n+1} = j \mid X_1 = i_1, X_2 = i_2, \ldots, X_n = i]$$

The value $P[X_{n+1} = j \mid X_n = i]$ is called the *transition probability* from state i to state j. Assuming our Markov chain can take on finitely many states, we can encode all possible transition probabilities in a transition diagram.



Here, let $|X(\Omega)| = 3$, and for simplicity let the range of possible states $X(\Omega) = \{1, 2, 3\}$. Then, the transition diagram is a directed graph with a vertex for every state and a directed edge $e = (i, j)$ for every non-zero transition probability. Let edge e have a weight equal to its transition probability, denoted p_{ij}.

Many of the classifications of Markov chains that we explored in class have a pattern that is only evident once drawing out the transition diagram. Using the following algorithms, we can search for these patterns computationally.

2. Adjacency Lists

Let V denote the set of all vertices in a graph and E the set of all edges. For some vertex $v \in V$, define the neighboring set of v as the set of all vertices $w \in V$ such that a directed edge (v, w) exists.

An adjacency list is a mapping between vertices and their neighboring sets. This is commonly [1] implemented using a hash-table, or a collection of key-value pairs. Given some edge $e = (v, w)$, we can use the hash-table to look up e in constant[1] time. By storing each neighboring vertex w as a tuple (w, p_{vw}) where p_{vw} is the weight of the edge e, we can look up the weight of e in constant time as well.

Key	**Value**
1	$\{(1, 0.2), (2, 0.6), (3, 0.2)\}$
2	$\{(1, 1.0)\}$
3	$\{(2, 1.0)\}$

Table 1: The adjacency list of the previous transition diagram.

The space complexity of the adjacency list is $O(V+E)$, meaning the storage requirement grows linearly with respect to the size of V and the size of E. Because our transition diagram has at most $|V|^2$ edges, we have an upper bound of $O(V^2)$.

We can then simulate a trajectory $X_1(\omega), X_2(\omega), \ldots$ of the Markov chain by traversing the adjacency list. Here, the algorithm will choose the state at time $k+1$ based on the weights of the edges stored in the neighboring set of the state at time k.

Algorithm 1 Generate $X_1(\omega), X_2(\omega), \ldots, X_n(\omega)$ for random $\omega \in \Omega$

Require: *adj_list*

```
function SIMULATE(v, n)
    trajectory ← [v]
    for k ← 1 to n do
        Sample (w, p_vw) from adj_list[v] based on weights p_vw
        v ← w
        trajectory.append(v)
    end for
    return trajectory
end function
```

Given $X_k = v$, the random variable X_{k+1} is distributed according to the weights $p_{v,w_1}, p_{v,w_2}, \ldots, p_{v,w_m}$. Let F_v denote the cumulative distribution function of X_{k+1}. Then,

$$F_v(w) = \begin{cases} 0 & \text{if } w < w_1 \\ \sum_{i=1}^{1} p_{v,w_i} & \text{if } w_1 \leq w < w_2 \\ \sum_{i=1}^{2} p_{v,w_i} & \text{if } w_2 \leq w < w_3 \\ \vdots & \vdots \\ 1 & \text{if } w_m \leq w \end{cases}.$$

Assuming we can generate a random number $u \in (0,1)$, we can sample from F_v using the piecewise inverse transform F_v^{-1} where $F_v^{-1}(u) = w_n$ if $\sum_{i=1}^{n} p_{v,w_i} < u \leq \sum_{i=1}^{n+1} p_{v,w_i}$.

[1] Average case only; $O(V)$ in the worst case due to possible hashing collisions.

For the algorithm to run according to its asymptotic lower bound $O(n)$, we must be able to calculate $\sum_{i=1}^{n} p_{v,w_i}$ in constant time.[2]

Each trajectory corresponds to a sequence of edges in the transition diagram. Specifically, for some trajectory $X_1(\omega) = v_1, X_2(\omega) = v_2, \ldots, X_n(\omega) = v_n$, there is a corresponding sequence of edges $e_1, e_2, \ldots, e_{n-1}$ such that $e_k = (v_k, v_{k+1})$. Denote $e_1, e_2, \ldots, e_{n-1}$ as a walk and $v_1, v_2, \ldots, v_n$ as the vertex sequence of the walk. If every vertex v_k is unique, then the walk is called a path. If every vertex v_k is unique except for the start and end vertices, the walk is called a cycle.

3. Communication Classes

A state i reaches state j (denoted $i \to j$) if there exists some n such that $P[X_n = j \mid X_0 = i] > 0$. By induction, we have:

$$\begin{aligned} P[X_n = j \mid X_0 = i] &= \sum_{k=1}^{|X(\Omega)|} P[X_n = j \mid X_{n-1} = k] P[X_{n-1} = k \mid X_0 = i] \\ &= \sum_{k=1}^{|X(\Omega)|} p_{kj} p_{ik}^{n-1} = p_{ij}^{n}. \end{aligned}$$

To determine if n exists computationally, we can use a depth-first search (DFS) algorithm to traverse all vertices $w \in V$ such that there exists a path $v, v_1, \ldots, w$.

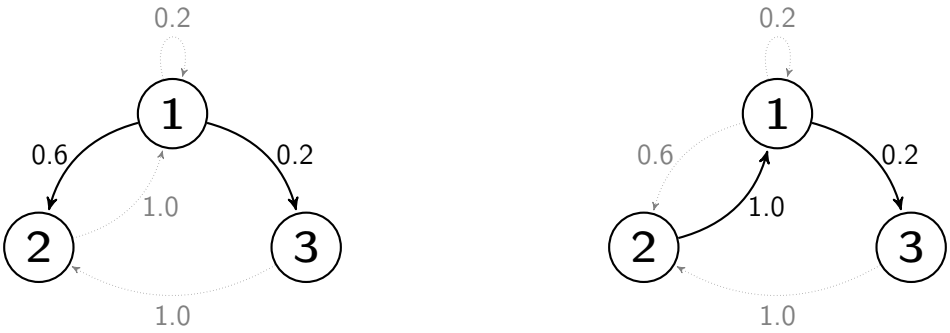
Algorithm 2 Depth-First Search

```
Require: adj_list
visited ← set()
function DFS(v)
    visited.append(v)
    for w in adj_list[v] do
        if w not in visited then
            DFS(w)
        end if
    end for
end function
```

The DFS algorithm begins at vertex v, iterates over its neighboring set, and recurs on a vertex w that was not visited before. This bounds the runtime of the DFS to the space of the adjacency list, being $O(V + E)$. The algorithm is called "depth" first because it explores the length of a given path first and exhausts the vertices in their neighboring sets after.

Let $E_i \subset E$ denote the set of edges traversed by the DFS call DFS(i), and let $V_i \subset V$ denote the set of visited vertices. The new graph $G_i \subset G$ is a directed, acyclic graph (DAG) containing all vertices j such that $i \to j$. See, for example, G_1 and G_2 of the previous diagram.

[2]This is possible using Walker's Alias Method after an O(V) upfront preprocessing cost [2].



States i and j communicate (denoted $i \leftrightarrow j$) if state i reaches state j and vice versa. We can determine this using two DFS calls: one beginning at state i and the other at state j. If i is in the visited set of j and vice versa, then $i \leftrightarrow j$.

Communication implies that there are paths $i, i_1, \ldots, j$ and $j, j_1, \ldots, i$ whose union $i, i_1, \ldots, j, j_1, \ldots, i$ creates a cycle[3]. Let the communication class $C[i]$ denote the set of all states j such that $i \leftrightarrow j$. Then, any vertex v in $\{i, i_1, \ldots, j, j_1, \ldots, i\}$ is also in $C[i]$.

To computationally find $C[i]$, we have to find the set of all cycles that contain state i. We can use our current approach to solve this in $|V|$ DFS calls, or $O(V \cdot (V + E))$ time.

4. Kosaraju's Algorithm

Kosaraju's algorithm was published in 1981 [3] as a way of discovering communication classes with a constant number of DFS calls. Therefore, it is linear with respect to the size of the adjacency list and improves on our previous approach.

Algorithm 3 Kosaraju's Algorithm for Communication Classes

Require: adj_list, visited as empty set

```
1:  function KOSARAJU(G)
2:      postorder ← [ ]
3:      for each v in adj_list do
4:          if v not in visited then
5:              postorder.extend(DFS(G, v, visited))
6:          end if
7:      end for
8:      visited ← set()
9:      classes ← list()
10:     while not postorder.empty() do
11:         v ← postorder.pop()
12:         if v not in visited then
13:             class ← set(DFS(G^T, v, visited)))
14:             classes.append(class)
15:         end if
16:     end while
17: end function
```

[3] If every vertex is unique. Otherwise, it's the union of multiple cycles.

Vertices v_a, v_d in a DAG G_i have an ancestor-descendant relationship if there exists a path $v_a, \ldots, v_d$. Here, we modify the DFS to return a history of vertices in the order their neighboring sets are exhausted, such that the set V_i is ordered into a sequence $v_1, v_2, \ldots, v_n$ where $a > d$ for all ancestor-descendant pairs $v_a, v_d \in V_i$. This is called a postorder traversal of G_i.

Define G^T as the transpose of the transition diagram G. This maps every edge $e = (u, v)$ to $e^T = (v, u)$. If there exists a path $v_a, \ldots, v_d$ in the DAG $G^T_{v_a}$ for some descendant v_d, then the inverse transpose of the path exists in G^T_i. This implies v_a, v_d form a cycle.

Since we're looking for paths of the form $v_a, \ldots, v_d$ in $G^T_{v_a}$, we must reverse the postorder traversal such that ancestors are visited *before* descendants. This is achieved using the pop method, which removes and returns the element at the end of an ordered list. Continuing this process until all vertices are visited determines every communication class using only $O(V + E)$ time.

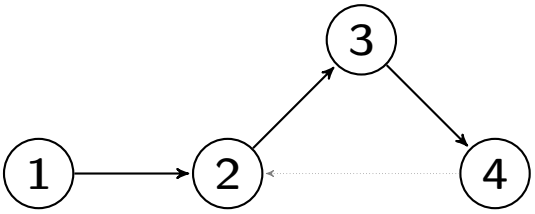


Figure 1: G_1

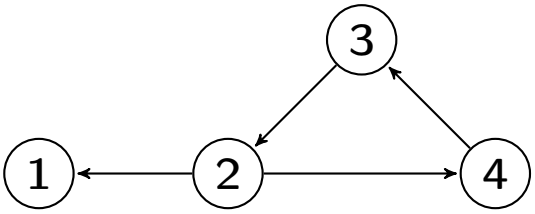


Figure 2: G^T

Example: G_1 has a postorder traversal $[4, 3, 2, 1]$. We pop the end vertex, 1. The visited set of G^T_1 is $\{1\}$. We pop the end vertex, 2. The visited set of G^T_2 is $\{2, 4, 3\}$. All vertices are visited; $C[1] = \{1\}$ and $C[2] = \{2, 4, 3\}$.

5. Recurrence and Transience

We can now define recurrence and transience based on the outdegree of a communication class. Let the outdegree of $C[i]$ be the number of edges $e = (u, v)$ such that $u \in C[i]$ and $v \notin C[i]$.

If the outdegree equals zero, the communication class is *recurrent*. If the outdegree is greater than zero, the communication class is *transient*. This classification can be found in $O(V + E)$ time. If the communication class $C[v] = \{v\}$ and the outdegree of $C[v]$ is zero, then the vertex v is *absorbing*.

Using computational graph algorithms, we can calculate many of the classifications of discrete-time Markov Chains that we went over in class, and do so at their most efficient possible time complexities. An extension of this is to explore the period of states, which, like recurrence, is common to states within the same communication class. One way to estimate the period is to take a sufficiently large sample of the walks of the form $i, \ldots, j$ and computing the GCD of their lengths.

References

[1] Guido van Rossum, "Graphs," Python.org, 1998. [Online]. Available: https://www.python.org/doc/essays/graphs/.

[2] Princeton University. *The Alias Method: Efficient Sampling with Many Discrete Outcomes*. 2013. Available online: https://lips.cs.princeton.edu/the-alias-method-efficient-sampling-with-many-discrete-outcomes/.

[3] Micha Sharir. A strong-connectivity algorithm and its applications to data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

I also used Microsoft Copilot to generate boilerplate code and interact with NetworkX/Matplotlib.

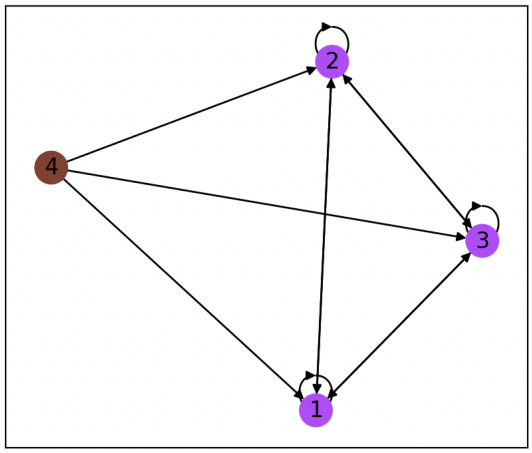
A. Additional Info

Big O notation is used to bound a function $f(x)$ with some other function $g(x)$ for sufficiently large x, up to some constant. Formally, $f(x)$ is $O(g(x))$ if there exists some $c \in \mathbb{R}$ and $x_0 \in \mathbb{R}$ such that

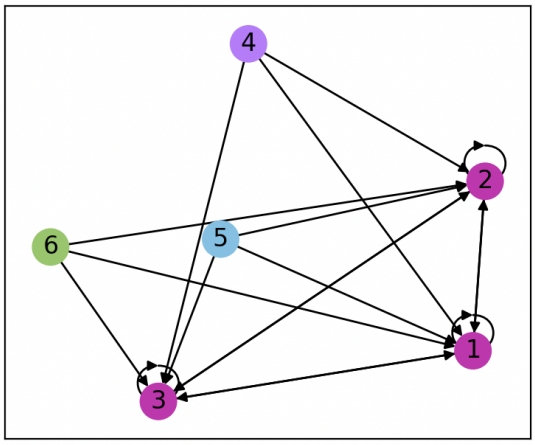
$$f(x) \leq c \cdot g(x) \quad \text{for all } x \geq x_0.$$

B. Source Code

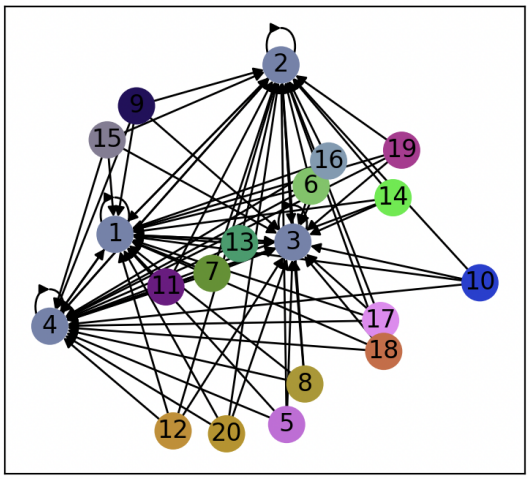
These visualizations of the Markov chains were made using NetworkX and Matplotlib libraries. I also assigned a color to each communication class to make them more visible. All algorithms are implemented and available at https://github.com/rshahid26/markov.



(a) 4 states



(b) 6 states



(c) Many states

Figure 3: Visualizations of Markov chains.