Machine Learning for Complex Networks

Prof. Dr. Ingo Scholtes

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

ingo.scholtes@uni-wuerzburg.de

Lecture 06
Random Walks and Flow Compression

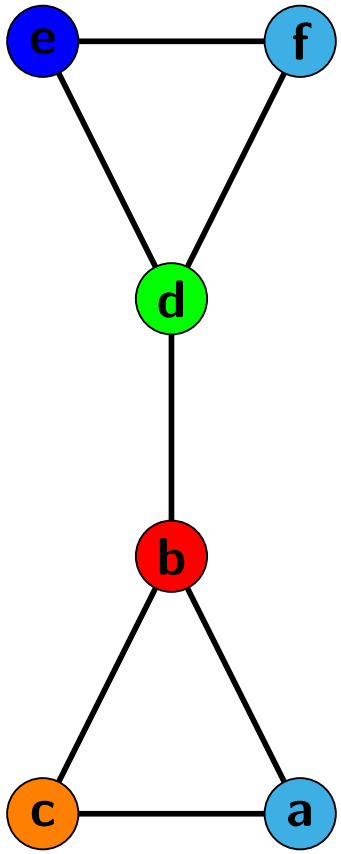
June 3, 2024

Notes:

- **Lecture L06:** Random Walks and Flow Compression 03.06.2024
- **Educational objective:** We introduce Markov chains models of random walks in a graph and study stationary visitation probabilities. We show how the compression of random walk sequences helps us to detect communities in networks.
  - Random Walks in Graphs
  - Stationary States of Random Walks
  - Compressing Random Walks
  - MapEquation and InfoMap

Motivation

- we used **description length minimization** to address **overfitting** in stochastic block model
- highlights relationship between **pattern recognition**, **model selection**, and **information theory**
- how else can we use **data compression** to detect **optimal communities**?
- we consider compression of node sequences generated by **random walk in a graph**



Notes:

- In the previous lecture we used description length minimization to address the problem of overfitting in community detection. In the stochastic block model, overfitting occurs if we naively search both for a number of blocks B and block assignment vector $\vec{z}$ that maximize likelihood. We found that the assignment of each node to a separate community trivially yields a model with likelihood one. In this case, the "learned" stochastic block matrix, which should ideally capture the community structure, simply corresponds to the adjacency matrix, i.e. we obtain a model that is as complex as our data set.
- To address this, we introduced the minimum description length principle. Its application to the stochastic block model led us to consider both the entropy of the ensemble and the complexity of the model description. Minimizing both at the same time allows us to infer parsimonious community structures, i.e. we can detect both the optimal assignment of blocks and the optimal number of blocks at the same time.
- This approach highlights interesting relations between data compression and model selection, which we will further explore today. Moving away from the inference of network ensembles, we consider a more direct and intuitive application of data compression, where we seek to compress the sequence of nodes traversed by a random walker in a graph.

Random walks in graphs

random walk model

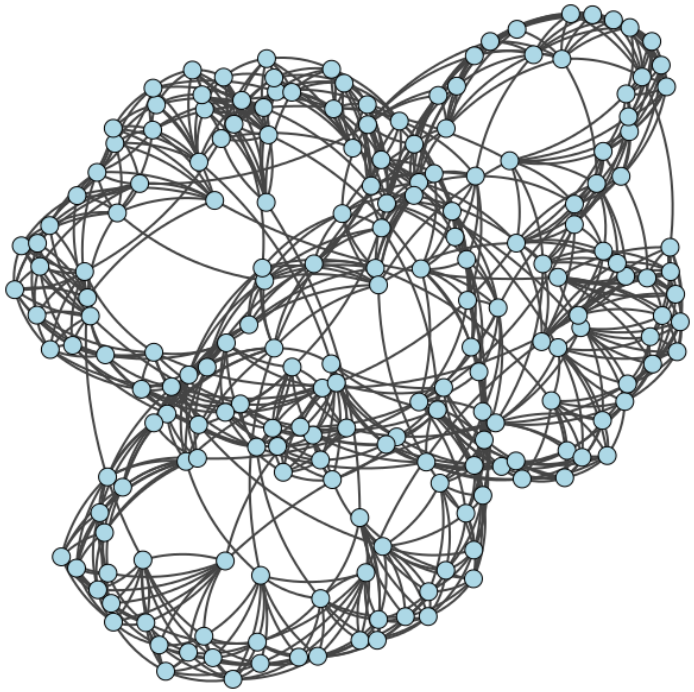
A random walk model is a stochastic model for walks through a **state space** Ω

- random variable $X_t \in \Omega$ assumes state of walker at time t
- sequence X_t is called a random walk
- time t and state space Ω can be discrete or continuous

- for random walk in graph $G = (V, E)$ we consider discrete state space $\Omega = V$, where **transitions occur across edges** E
- important foundation to study **diffusion processes** and define **centrality measures**

  → Statistical Network Analysis, L10 – L13
- basis for **machine learning in graphs**

  → Lecture 10: Walk-based Embedding



trajectory of random walker in a complex network

Notes:

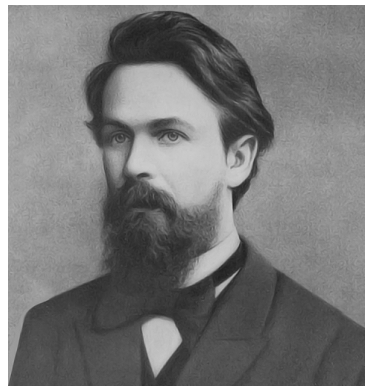
- We first introduce a simple class of probabilistic models for **random walks in a graph**. Such models are very important in network analysis and graph learning, where we can use them to model diffusion processes, detect communities, assess the centrality of nodes, or generate embeddings of nodes in a vector space.
- A random walk can be viewed as a probabilistic generative model for a (possibly infinite) sequence of random variables $X_t \in \Omega$, where X_t is the state of the random walker at time t. The discrete state space Ω is given by the vertices V of a graph $G = (V, E)$, i.e. at each time t the random walker can reside at exactly one node $v \in V$. In the network context, each random realization of this sequence X_t is a walk in G (i.e. a path where each node can appear multiple times).
- This simple model provides us with a simple method to calculate, not only the probability of paths, but also the probability that our random walker resides at any given node at any given point in time.

Random walks as Markov chains

- consider a **discrete-time** random walk on finite and **discrete state space** Ω where $X_t \in \Omega$ is the state at time t

- at time t **transition probability** of random walk to s_{t+1} is

$$P(X_{t+1} = s_{t+1} | X_t = s_t, X_{t-1} = s_{t-1}, \ldots X_0 = s_0)$$



Andrey Markov

1856 – 1922

- process is **memoryless** (has the Markov property) iff

$$P(X_{t+1} = s_{t+1} | X_t = s_t) = P(X_{t+1} = s_{t+1} | X_t = s_t, X_{t-1} = s_{t-1}, \ldots X_0 = s_0)$$

for all sequences of states $s_0, \ldots, s_{t-1} \in \Omega$

observations

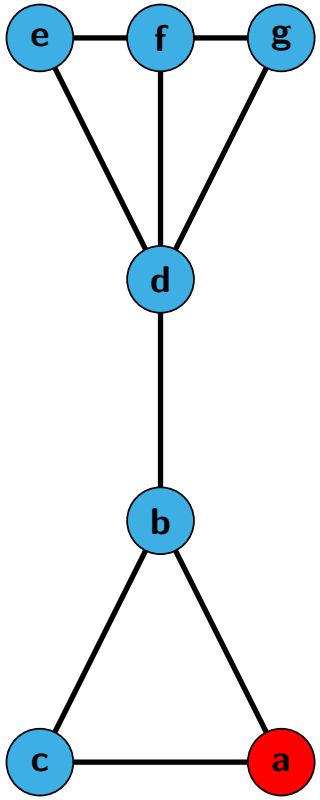
- future state X_{t+1} only depends on current state X_t
- discrete-time/discrete-state stochastic process with Markov property is called **Markov chain**
- transition probabilities between i and j can be given as **entries T_{ij} of matrix T**

image credit: Wikimedia Commons, public domain

Notes:

- But how exactly do we model how the random walker moves between nodes? We assume a simple probabilistic model where each next state is randomly chosen from the set of possible next states. That is, at each time t the choice where to move next is made by means of a random experiment, where we assign transition probabilities to all subsequent states s_{t+1}.
- We can think of numerous ways to assign those probabilities, each capturing a different "model" of the underlying process. In general, the probability of a transition to the next state can depend on the whole history of the process, i.e. where the process has been in each of the prior steps.
- A particularly simple (and well-studied) class of random walk models are those which are "memoryless". This means we assume that the probabilities for transitions to the next state only depend on the current state. In other words we assume that – as soon as a random walker arrives at state s_t – it has "forgotten" where it was before. Such a random walk is a primary example for a so-called **memoryless Markov process**, i.e. a class of processes where the future trajectory of the process is independent of the past. In this case, we can simply describe the transition probability of the process by a simple condition probability.
- If both time and state space are discrete we obtain a **Markov chain**, named after the Russian mathematician Andrey Markov who studied such processes in the late 19th century.

Adjacency matrices of networks



random walker with $X_0 = a$

$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

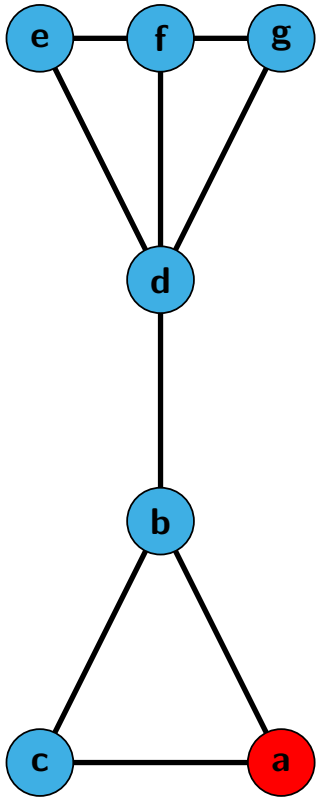
for matrix entry A_{ij}, i refers to row and j refers to column

how can we define entries of **transition matrix T**?

Notes:

- Since the probability to transition to a state j at time $t+1$ only depends on which state i the process is in at time t, we can define a transition matrix $\mathbf{T}$ with entries T_{ij} being the transition probabilities.
- How can we define the transition matrix of such a random walk process in a graph, where the states are the nodes in which the random walker currently resides?
- Consider how we have defined the adjacency matrix of a network. We used the (usual) notation where for entry A_{ij}, index i refers to the row and j refers to the column.
- Let us now consider a random walker in node (i.e. state) a as illustrated above. The two one entries A_{ab} and A_{ac} correspond to the two possible next states s_{t+1} of a random walker that is in state a at time t.

Transition matrix of a random walk



random walker with $X_0 = a$

$$\mathbf{T} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix} \end{array}$$

T_{ij} captures **probability of walker** residing in node i to move to node j

we can **define entries of a transition matrix** as

$$T_{ij} := A_{ij} \cdot \left(\sum_{k \in V} A_{ik} \right)^{-1} = \frac{A_{ij}}{d_{out}(i)}$$

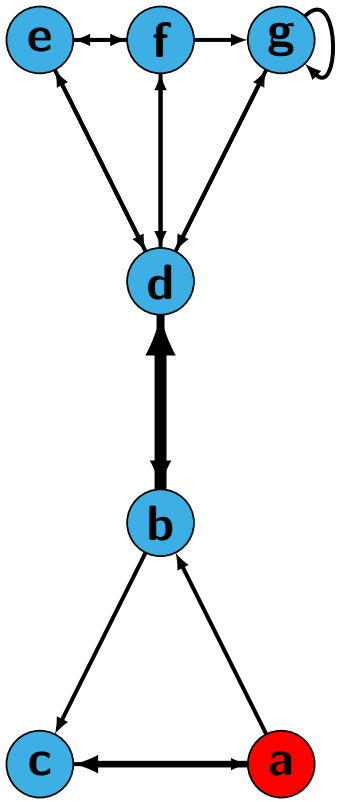
Notes:

- In absence of any further knowledge about the random walker it is reasonable to make a maximum entropy assumption, i.e. we assign the same probability for the transition to each of our neighbours.
- For each entry A_{ij} in the adjacency matrix of a directed network, we can take the inverse of the out-degree of node i, which counts the number of possible next states (i.e. nodes) that we can transition to. We obtain a transition matrix for a random walk process where in each step the next node is chosen uniformly at random from the set of all possible next nodes.
- In other words: we define a transition matrix $\mathbf{T}$ by dividing the entries of the adjacency matrix $\mathbf{A}$ of a network by the row sums, i.e.

$$T_{ij} = \frac{A_{ij}}{\sum_k A_{ik}}$$

- Note that in this notation, i designates the row and j designates the column of the matrix, i.e. in the example above we have $T_{ab} = \frac{1}{2}$ while $T_{ba} = \frac{1}{3}$.
- This notation implies that all rows of our transition matrix sum up to one (i.e. we have a so-called row-stochastic or right-stochastic matrix, but more on this later).

Biased random walks in weighted networks



$$\mathbf{T} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & \frac{1}{4} & \frac{3}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{8} & \frac{7}{8} & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{4}{7} & 0 & 0 & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix} \end{array}$$

in weighted networks we use weights to **bias transition probabilities**, e.g.

$$T_{ij} := A_{ij} \cdot \left(\sum_{k \in V} A_{ik} \right)^{-1} = \frac{w(i,j)}{\sum_{k \in V} w(i,k)}$$

Notes:

- What if we have more information about the random walk process?
- For weighted networks, we can consider non-uniform, **biased transition probabilities** to move to different neighbors. Depending on what we want to model, we can think of many different ways to do this. Here we have used the simplest possible bias in transition probabilities: a bias that is proportional to weights.
- in the matrix above, the entries $T_{ab} = \frac{1}{4}$ and $T_{ac} = \frac{3}{4}$ are proportional to link weights $w(a, b) = 1$ and $w(a, c) = 3$
- This implies that we do not preserve the absolute weights of links, we normalize the weights for each node individually to obtain proper transition probabilities. Specifically, we can scale the weights of the outgoing links of each node by a constant and still obtain the same transition matrix (because we normalize with the sum of out-weights for each node individually). This implies that different weighted networks give rise to the same transition matrix!

Stochastic matrices

- row sums in transition matrix $\mathbf{T}$ are one, i.e.

$$\sum_{j \in V} T_{ij} = 1 (\forall i \in V)$$

- such a matrix is called **row-stochastic** or **right-stochastic**

$$\mathbf{T} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix} \end{array}$$

- let $\pi = (\pi_i)_{i \in V}$ be a **stochastic row vector**, i.e. $\sum_i \pi_i = 1$

- $\pi \cdot \mathbf{T}$ exists and is again a stochastic row vector

example

$$\pi = (1, 0, 0, 0, 0, 0, 0)$$

$$\pi \cdot \mathbf{T} = \left(0, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0\right)$$

Notes:

- In both cases (weighted and unweighted networks) we obtain a matrix where each row is a probability mass function. Such a matrix is called a **row- or right-stochastic** matrix.
- Why right-stochastic? Because if we multiply $\mathbf{T}$ from the **right** with a stochastic row vector v, i.e. $v \cdot \mathbf{T}$, then the result is again a stochastic row vector, i.e. a vector whose entries sum to one.
- Alternatively, we can also consider **column- or left stochastic** matrices, but then we have to consider column vectors and we need to multiply $\mathbf{T}$ from the left, i.e. we consider $v = (1, 0, 0, 0, 0, 0, 0)^T$ and $\mathbf{P} \cdot v$ returns a stochastic column vector.
- For a network with n nodes, let us take a closer look at a stochastic vector $\pi \in \mathbb{R}^n$ and transition matrix $\mathbf{T} \in \mathbb{R}^{n \times n}$. We can interpret π as a vector that describes the probability of a random walk to reside in particular nodes.
- Let us now consider a canonical unit vector π that captures the state of a random walker residing in a given node with probability one (see above). What do we get for $\pi \cdot \mathbf{T}$?
- We obtain a vector that captures the probabilities of a random walker to be in a particular node after a single step of the random walk (assuming that it started in the node captured by π).

Visitation probabilities

- let $\pi^{(t)} = \left(\pi_1^{(t)}, \pi_2^{(t)}, \ldots\right)$ be **visitation probabilities** at time t
- we call $\pi^{(0)}$ the **initial distribution** of a random walk

example

$$\pi^{(0)} = (1,0,0,0,0,0,0)$$
$$\pi^{(1)} = \left(0, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0\right) = \pi^{(0)} \cdot \mathbf{T}$$
$$\pi^{(2)} = \left(\frac{5}{12}, \frac{1}{4}, \frac{1}{6}, \frac{1}{6}, 0, 0, 0\right)$$
$$= \pi^{(1)} \cdot \mathbf{T} = \pi^{(0)} \cdot \mathbf{T}^2$$

$$\mathbf{T} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix} \end{array}$$

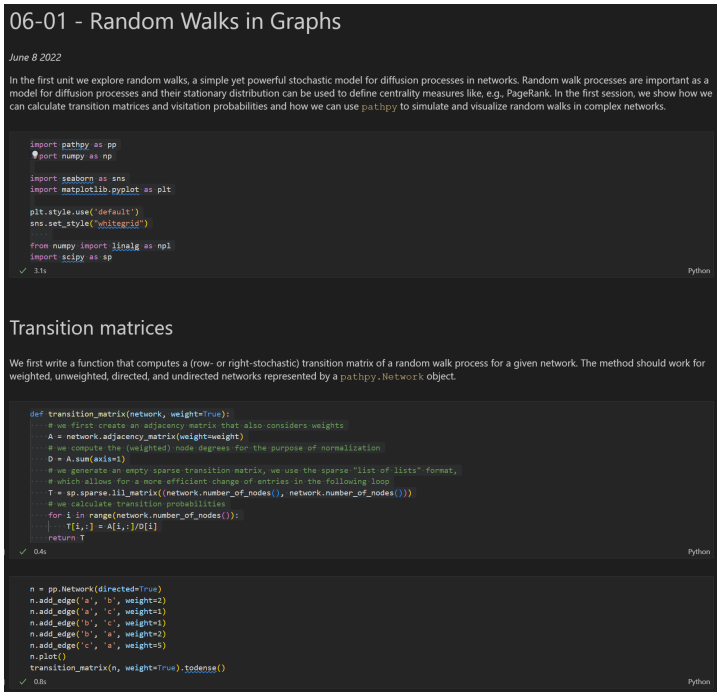
using initial distribution $\pi^{(0)}$ and transition matrix $\mathbf{T}$ **visitation probabilities at time t** are given as $\pi^{(t)} = \pi^{(0)} \cdot \mathbf{T}^t$

Notes:

- We can encode the initial state of a random walk process in a stochastic vector $\pi^{(0)}$, which we refer to as the **initial distribution**. We can multiply this vector with the transition matrix and call the result $\pi^{(1)}$.
- Comparing $\pi^{(0)}$ with $\pi^{(1)}$, we observe that the initial probability 1 to be at node a gets "distributed" to probabilities $\frac{1}{2}$ at nodes b and c to whom node a is connected. This is easy to understand, because (i) the transition matrix $\mathbf{T}$ captures the topology of links in the network, and (ii) $\mathbf{T}$ is a right-stochastic matrix, so we again obtain a probability distribution.
- The same applies if we repeatedly multiple the transition matrix with a stochastic vector. The result of k repeated multiplications corresponds to a single multiplication with the k-th power of the transition matrix. That is, for a given transition matrix and a given initial state we can directly calculate the probabilities that the random walker is at any given node after k steps. This is due to the definition of matrix multiplication, where the entries of the k-th power of an adjacency (or transition) matrix capture all paths of exactly length k.
- In the example above, for entries $\pi_b^{(1)}$ and $\pi_c^{(1)}$ we have a difference of $\frac{1}{2}$ compared to $\pi_b^{(0)}$ and $\pi_c^{(0)}$, while for $\pi_a^{(1)}$ we have a difference of 1. All other entries remain unchanged.
- How can we quantify these changes and how do the changes in probability evolve over time?

Practice Session

- we calculate **transition matrices of random walks in directed and undirected networks**
- we use matrix powers to calculate **visitation probabilities of nodes**
- we **simulate and visualize random walks** using `pathpy`



06-01 - Random Walks in Graphs

June 8 2022

In the first unit we explore random walks, a simple yet powerful stochastic model for diffusion processes in networks. Random walk processes are important as a model for diffusion processes and their stationary distribution can be used to define centrality measures like, e.g., PageRank. In the first session, we show how we can calculate transition matrices and visitation probabilities and how we can use `pathpy` to simulate and visualize random walks in complex networks.

```
import pathpy as pp
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

plt.style.use('default')
sns.set_style("whitegrid")

from numpy import linalg as npl
import scipy as sp
```

✓ 3.1s Python

Transition matrices

We first write a function that computes a (row- or right-stochastic) transition matrix of a random walk process for a given network. The method should work for weighted, unweighted, directed, and undirected networks represented by a `pathpy.Network` object.

```
def transition_matrix(network, weight=True):
    # we first create an adjacency matrix that also considers weights
    A = network.adjacency_matrix(weight=weight)
    # we compute the (weighted) node degrees for the purpose of normalization
    D = A.sum(axis=1)
    # we generate an empty sparse transition matrix, we use the sparse "list of lists" format,
    # which allows for a more efficient change of entries in the following loop
    T = sp.sparse.lil_matrix((network.number_of_nodes(), network.number_of_nodes()))
    # we calculate transition probabilities
    for i in range(network.number_of_nodes()):
        T[i,:] = A[i,:]/D[i]
    return T
```

✓ 0.4s Python

```
n = pp.Network(directed=True)
n.add_edge('a', 'b', weight=2)
n.add_edge('a', 'c', weight=1)
n.add_edge('b', 'c', weight=1)
n.add_edge('b', 'a', weight=2)
n.add_edge('c', 'a', weight=5)
n.plot()
transition_matrix(n, weight=True).todense()
```

✓ 0.8s Python

practice session

see notebook 06-01 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

- In the first practice session, we calculate the transition matrix of a random walk in a graph and use matrix powers to calculate the evolution of visitation probabilities over time.
- We further use `pathpy` to simulate and visualize random walks in graphs.

Total variation distance

- we can view **evolution of visitation probabilities** as **diffusion process**

total variation distance

to measure difference between π and π' we define the **total variation distance** as

$$\delta\left(\pi, \pi'\right) := \frac{1}{2} \sum_i \left|\pi_i - \pi'_i\right|$$

- in our example network, we observe

$$\delta\left(\pi^{(t)}, \pi^{(t-1)}\right) \to 0$$

for $t \to \infty$

example: $t = 15$ vs. $t = 16$

$$\pi^{(15)} \approx (0.124, 0.178, 0.124, 0.212, 0.103, 0.153)$$

$$\pi^{(16)} \approx (0.122, 0.177, 0.122, 0.214, 0.104, 0.156)$$

total variation distance in example

$$\delta\left(\pi^{(2)}, \pi^{(1)}\right) \approx 0.583$$

$$\delta\left(\pi^{(16)}, \pi^{(15)}\right) \approx 0.007$$

Notes:

- In the example considered in the practice session, we observe that the node visitation probabilities cease to change for sufficiently large times t. The evolution of visitation probabilities can be viewed as a diffusion process that –under certain conditions– reaches a stationary state.
- To assess this quantitatively, we can calculate a distance measure between two probability mass functions. A popular measure is the so-called **total variation distance**, which sums the component-wise (absolute) differences between two stochastic vectors. The total variation distance (TVD) is at most 1, since the sum of the differences for each dimension is at most two, and we divide the sum by two.
- We can now study how the total variation distance between two consecutive visitation probability vectors $\pi^{(t)}$ and $\pi^{(t+1)}$ evolves for increasing t. For our simple example we find that the difference decreases over time (in this case it actually converges to zero). This means that – if we allow the random walk process to run long enough – the visitation probabilities will not change any more.
- As an example, consider that you are searching for a drunk friend walking in Würzburg. Using the initial location as initial distribution, you can model the walk of your friend as a random walk through the streets of Würzburg (assuming he or she does not leave the city). After a long enough time you can calculate (stable) probabilities to find him or her at any of the possible locations in Würzburg. You could then search at those locations that have the the highest visitation probability. Similar ideas have been used in real search and rescue operations, e.g. in the search for missing ships.

Stationary distribution of random walks

- in our example $\pi^{(t)}$ ceases to change, i.e. process reaches a **stationary distribution** $\pi := \pi^{(\infty)}$ such that

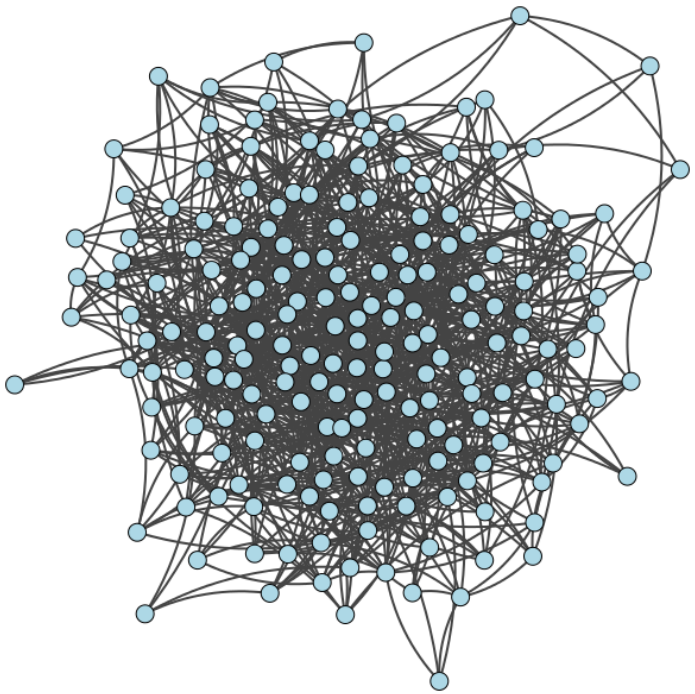
$$\pi = \pi \cdot \mathbf{T}$$

- this is an **eigenvalue problem** of the form

$$\pi \cdot v = \pi \cdot \mathbf{T}$$

with $v = 1$

- we can calculate stationary distribution as **left eigenvector of transition matrix corresponding to eigenvalue** $v = 1$



convergence to stationary distribution in an example network

Notes:

- In cases where the visitation probability $\pi^{(t)}$ ceases to change after long enough time, we call the limiting distribution $\pi := \pi^{(\infty)}$ the **stationary distribution** of the random walk process. We can easily calculate this stationary distribution by solving an eigenvector problem for the eigenvalue one of the transition matrix.
- Note that – since the transition matrix T is row-stochastic, i.e. rows sum up to 1 – the eigenvalue $v = 1$ is at the same time the largest eigenvalue of matrix $\mathbf{T}$.
- Also, here we must use the **left eigenvector**, because $\mathbf{T}$ is a row- or right-stochastic matrix. If we were to define $\mathbf{T}$ as a column- or left-stochastic matrix, we would need to consider right eigenvalues, i.e. we would need to solve the eigenvalue problem:

$$\mathbf{T} \cdot \pi = \pi \cdot v$$

- In general, the eigenvalue equation above can have multiple solutions for $v = 1$, i.e. we can have several vectors π that all represent stationary distributions of the random walk process (for different initial distributions $\pi^{(0)}$. We will see that the question whether the eigenvalue 1 occurs only once in the sequence of all eigenvalues of matrix $\mathbf{T}$ has an interesting graph-theoretic analogy. Similarly, there can be the case that there is no solution to the eigenvector equation, in which case visitation probabilities do not converge even for $t \to \infty$.

Markov chain convergence theorem

Markov chain convergence theorem

Let $\mathbf{T}$ be an **irreducible and aperiodic transition matrix** and let $\pi^{(0)}$ be an **arbitrary initial distribution**.

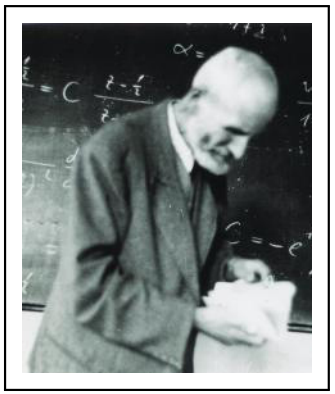
For $\pi^{(t)} := \pi^{(0)} \cdot \mathbf{T}^t$ there exists **exactly one** stationary distribution π such that

$$\delta(\pi^{(t)}, \pi) \to 0 \qquad (t \to \infty)$$

Follows from **Perron-Frobenius theorem** for real-square matrices with positive entries

interpretation in graphs → cf. Statistical Network Analysis

- $\mathbf{T}$ irreducible $\leftrightarrow$ strongly connected graph
- $\mathbf{T}$ aperiodic $\leftrightarrow \nexists k > 1$ that divides length of every cycle in graph
- if conditions of theorem are satisfied, π is **independent from initial distribution**
- over time, random walker "forgets" where it started



Oskar Perron
1880 – 1975

image credit: Konrad Jacobs, Wikimedia Commons, CC BY-SA 2.0

Georg Frobenius
1849 – 1917

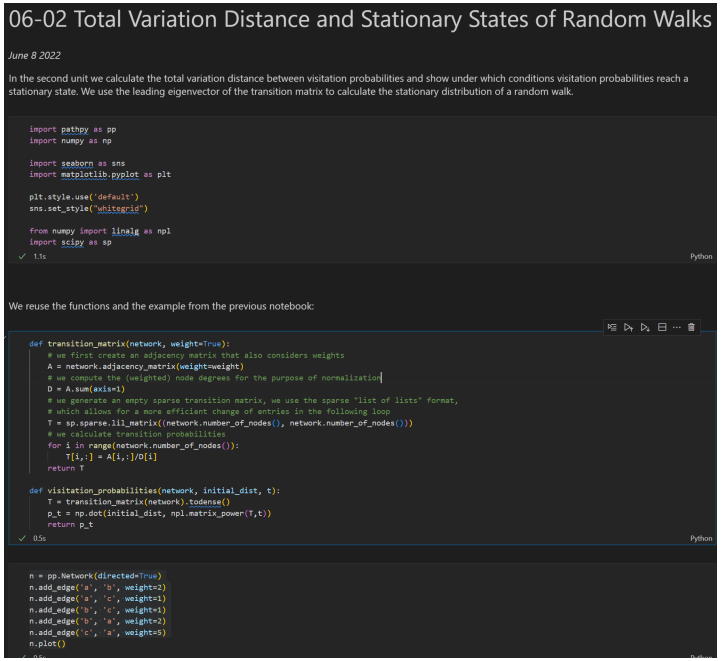
image credit: Oberwolfach Photo Collection, public domain

Notes:

- The Markov chain convergence theorem clarifies the conditions under which a Markov chain has a unique stationary state. It follows from the Perron-Frobenius theorem for non-negative (i.e. all values are either zero or positive) and irreducible matrices where all states have period one, see → O Perron, 1907 and → G Frobenius, 1912
- In the context of graphs, the property of **irreducibility** refers to the existence of a singular (strongly) connected component. Clearly, if a graph has more than one (strongly) connected component, random walks starting in different components reach different stationary states.
- **Aperiodicity** refers to the fact that there cannot be "periods" in the length of cycles, i.e. walks that start in a node and return to that node. If we can only return to a node at specific multiples of t, we never reach a stationary state. This is explained in more detail in the script *Statistical Network Analysis*. We can ensure aperiodic transition matrices by including self-loops, which leads to a *lazy* random walk.
- We omit the proof of the theorem, but a nice coverage of Markov chains with easy to follow proofs (e.g. of the Markov chain convergence theorem), illustrative examples and applications can be found in → O Häggström, 2002
- An important implication of the theorem is that, if the assumptions of the theorem are satisfied, the stationary distribution—since it is unique— is independent of the initial distribution. This implies that the random walker, over time, forgets in which node it started. This is a result of the memoryless/Markov property of the process, by which we "destroy information" on our origin in each step of the process.

Practice Session

- we calculate the **total variation distance** between stochastic vectors
- we use eigenvectors to calculate **stationary node visitation probabilities** in networks
- we calculate stationary visitation probabilities in graphs with **multiple connected components**



06-02 Total Variation Distance and Stationary States of Random Walks

June 8 2022

In the second unit we calculate the total variation distance between visitation probabilities and show under which conditions visitation probabilities reach a stationary state. We use the leading eigenvector of the transition matrix to calculate the stationary distribution of a random walk.

```
import pathpy as pp
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

plt.style.use('default')
sns.set_style("whitegrid")

from numpy import linalg as npl
import scipy as sp
```

✓ 1.1s Python

We reuse the functions and the example from the previous notebook:

```
def transition_matrix(network, weight=True):
    # we first create an adjacency matrix that also considers weights
    A = network.adjacency_matrix(weight=weight)
    # we compute the (weighted) node degrees for the purpose of normalization
    D = A.sum(axis=1)
    # we generate an empty sparse transition matrix, we use the sparse "list of lists" format,
    # which allows for a more efficient change of entries in the following loop
    T = sp.sparse.lil_matrix((network.number_of_nodes(), network.number_of_nodes()))
    # we calculate transition probabilities
    for i in range(network.number_of_nodes()):
        T[i,:] = A[i,:]/D[i]
    return T

def visitation_probabilities(network, initial_dist, t):
    T = transition_matrix(network).todense()
    p_t = np.dot(initial_dist, npl.matrix_power(T,t))
    return p_t
```

✓ 0.5s Python

```
n = pp.Network(directed=True)
n.add_edge('a', 'b', weight=2)
n.add_edge('a', 'c', weight=1)
n.add_edge('b', 'c', weight=1)
n.add_edge('b', 'a', weight=2)
n.add_edge('c', 'a', weight=5)
n.plot()
```

practice session

see notebooks 06-02 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

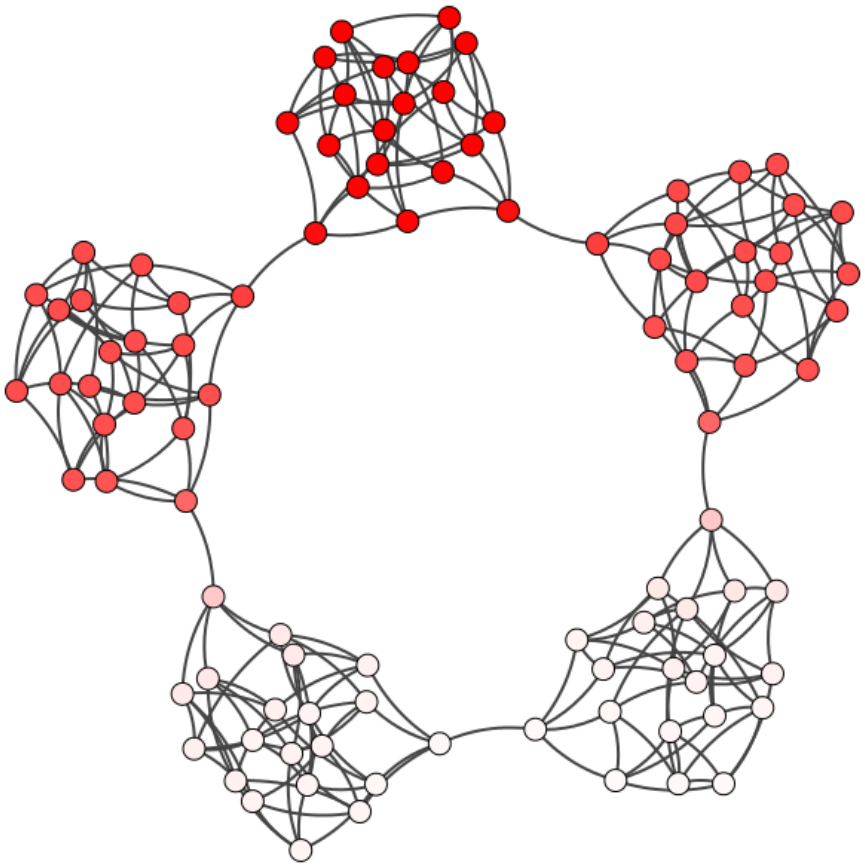
- In the practice session, we experimentally test the conditions under which the Markov chain convergence theorem holds, i.e. we test for the existence and uniqueness of the stationary distribution in networks with different topologies.
- Our results show that we can use the sequence of eigenvalues of a transition matrix to make statements about the topology of a network, namely how many (strongly) connected components it contains. This is related to the nmultiplicity of zero in the eigenvalue sequence of the graph Laplacian, which corresponds to the number of connected components in an undirected graph.

Random walks and community detection

- consider **random walk** in a graph, i.e. **ordered sequence of visited nodes**
- partition of nodes into “natural communities” allows to **compress random walk description**

example

Trajectory of **random walker in graph with five communities**. Colors indicate probability that node is visited after t steps.



Martin Rosvall

born 1978

image credit: Twitter/m_rosvall

Carl T. Bergstrom

born 1971

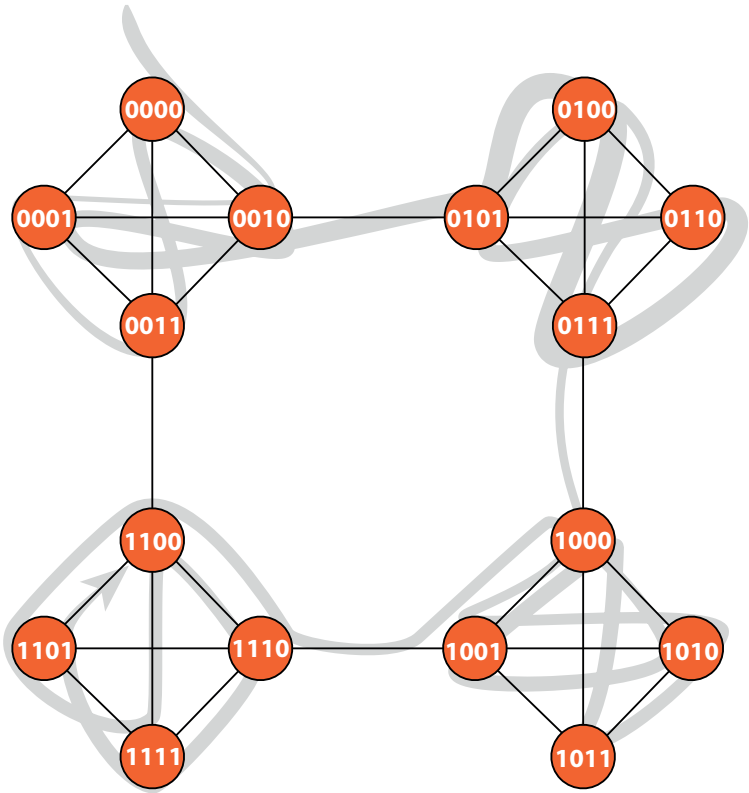
image credit: University of Washington

Notes:

- Having covered the basics of random walks in graphs, let us now consider how a random walk process evolves in the undirected graph shown on the right. This graph has a clear cluster pattern with five communities that are connected in a ring lattice topology.
- If we monitor the sequence of nodes traversed by a random process, we see that the random walker is continuously trapped within one community, before moving to another community, where it is again trapped for a long time. This is due to the high density of links within communities, as opposed to the low link density across communities. In other words, our density-based definition of communities in a graph naturally translates to a pattern in the sequence of nodes visited by a random walker.
- Taking an information-theoretic point of view leads us to an elegant community detection algorithm that is based on the minimum description length principle. The idea is to consider an encoding that stores the sequence of nodes that are visited by a random walker in the network above. This sequence exhibits a pattern that is due to the community structure.
- If we use a coding of nodes that utilizes the pattern that is due to the natural community structures of the graph, we can store the sequence of traversed nodes more efficiently. This is the basis for the flow compression algorithm InfoMap, which was introduced in the very readable and didactical work → M Rosvall, CT Bergstrom, 2008

Description length of random walkers

- consider **random walk** in undirected graph with **binary encoded nodes**
  - start in node 0000
  - in each step move to neighbor with probability T_{ij}
- how many bits are needed to store the trajectory of a random walk with 32 steps?
- we can store any sequence using **4 bits per random walk step**
- can we compress the sequence?



0000|0010|0001|0011|0001|0010|0101|...

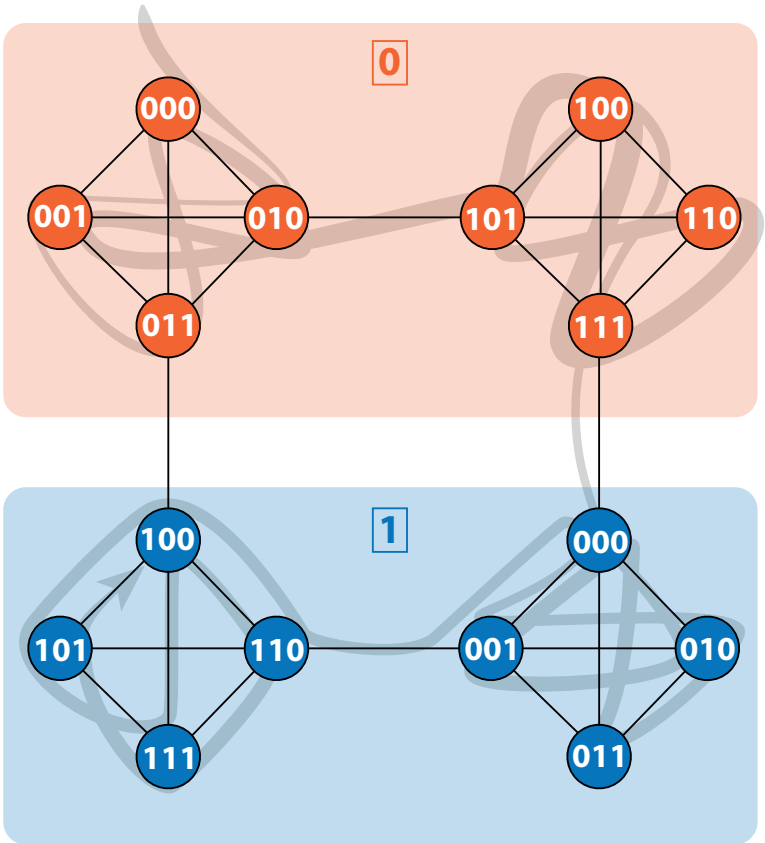
$$\underbrace{32 \cdot 4 \text{ bits}}_{\text{switch between nodes}} = 128 \text{ bits}$$

Notes:

- To motivate this algorithm, consider an undirected and unweighted toy example. We further consider a particular trajectory of a random walk (i.e. a sequence of visited nodes) with 32 steps (indicated in grey) generated by a random walk model. We note that, to simplify the discussion, this sequence has been generated such that each node appears with the same frequency, i.e. in this particular example we cannot encode the sequence more efficiently if we simply consider non-uniform occurrences of symbols (e.g. through Huffman coding).
- How many bits do we need to store this sequence of nodes? This may sound like a trivial question: We have a total of 16 nodes, which means we need to use 4 bits to encode the nodes. A sequence of 32 node visitations requires $32 \cdot 4 = 128$ bits, i.e. we need 4 bits to encode each step of the random walk.

Flow compression: underfitting

- consider mapping C_2 of nodes to **two communities** 0 (red) and 1 (blue)
- use **hierarchical coding scheme** where nodes are identified relative to their community
  - 1 bit prefix identifies community
  - 3 bit suffix identifies node **within** community
- omission of redundant community prefixes allows to **compress flow**



0000|010|001|011|001|010|...**1**000|010|001|011|...

$$\underbrace{2 \cdot 1 \text{ bit}}_{\text{switch between clusters}} + \underbrace{32 \cdot 3 \text{ bits}}_{\text{switch between nodes}} = 98 \text{ bits}$$

example

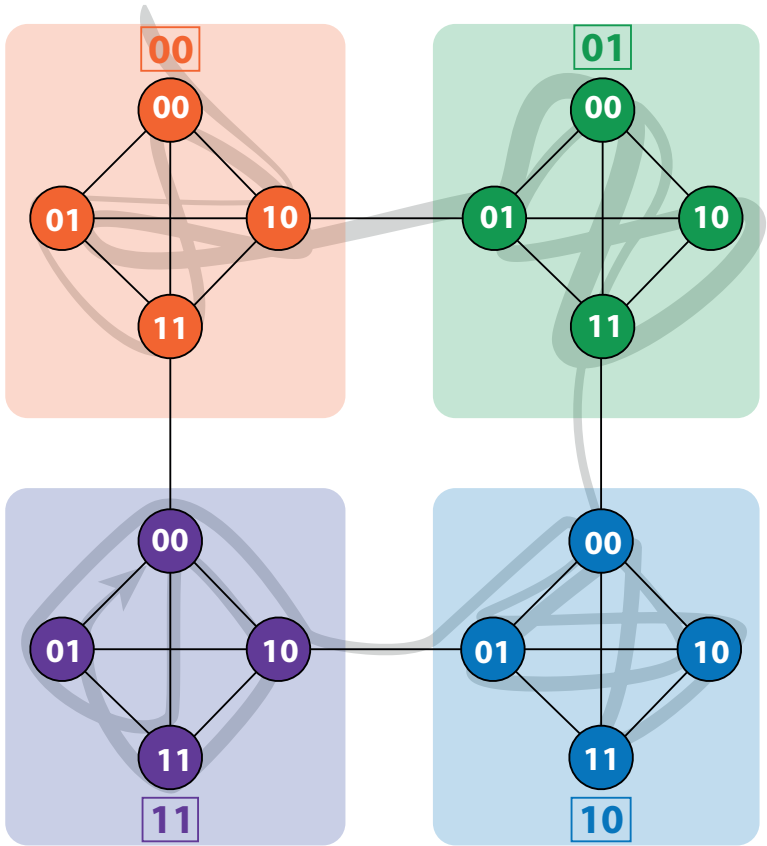
sequence of 32 traversed nodes can be encoded with 3.0625 bits per random walk step

Notes:

- We can **use the community structure of the network to compress our sequence**. For this, we use a mapping C_2 of nodes to two communities, i.e. the cluster of node i is given by $C_2[i]$.
- We can turn this into a **hierarchical coding scheme**, where we use a prefix/suffix coding to identify nodes (suffix) relative to the community (prefix) to which they are assigned. For two communities, we use the first (most significant) bit to identify the community to which a node belongs. Since each of these two communities contains eight nodes, we can use the three remaining (least significant) bits to encode which of the eight nodes within each of the communities we want to address.
- If we assign the two communities in a meaningful way, i.e. community 1 consists of the top eight, and community 2 consists of the bottom 8 nodes, we obtain a high level of redundancy in terms of the community codes that we can use for compression. We could denote the community once, and implicitly use node identifiers that are relative to the current community.
- Due to the effect of the network topology on the random walker, for a natural community partition, we expect many subsequent transitions within the same community since, by definition, each node has more links to nodes within the same community than to other communities. This translates to a more efficient coding of the sequence, i.e. in the example above we can use it to compress the sequence to 98 bits or $3.0625 < 4$ bits per random walk step.
- Note: In the simplified example above, we have used vertical bars to explicitly distinguish between community and node labels. Without those vertical bars, the

Flow compression: “optimal” clusters

- consider mapping C_4 of nodes to **four communitoes** 0, 1, 2, and 3
- use **hierarchical coding scheme** where nodes are identified relative to their community
  - 2 bits prefix identifies community
  - 2 bits suffix identifies node
- C_4 better matches “natural” communities and thus yields higher **compression**



example

sequence of 32 traversed nodes can be encoded with 2.25 bits per random walk step

00|00|10|01|11|01|10|... 01|00|01|10|11|00|... 10|00|...

$$\underbrace{4 \cdot 2 \text{ bit}}_{\text{switch between clusters}} + \underbrace{32 \cdot 2 \text{ bits}}_{\text{switch between nodes}} = 72 \text{ bits}$$

Notes:

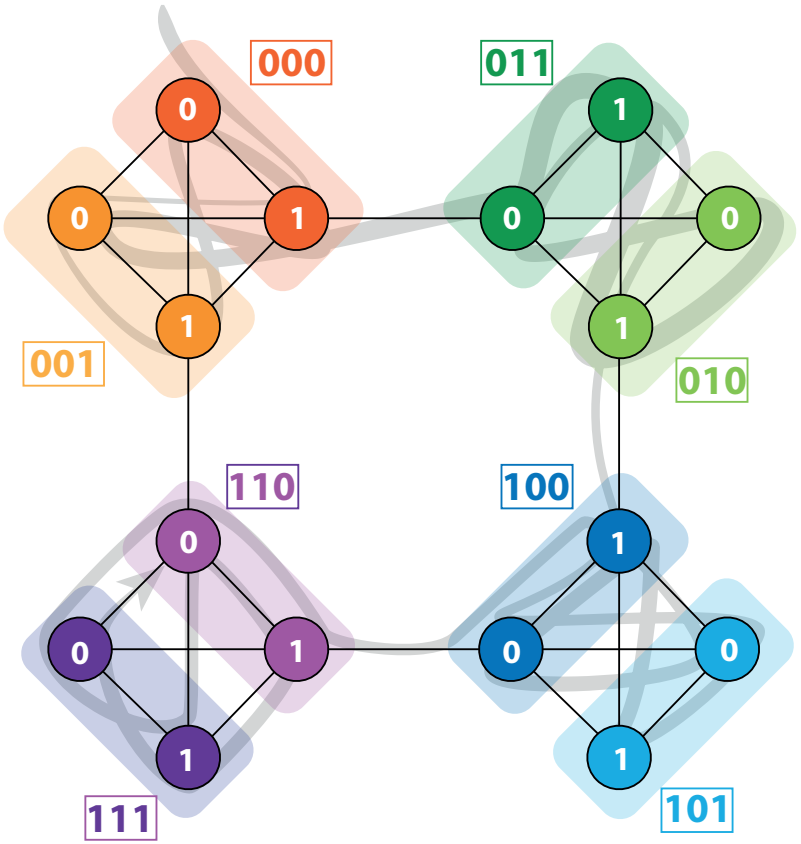
- The example from the previous slide actually corresponds to an “underfitting” of the community structure in our network, as it does not capture the four “natural” communities in our example network. From a compression point of view, a mapping to two communities uses the redundancy of community prefixes, since the first 16 visited nodes are in the same community. However, there is unused redundancy, because the first eight nodes are in a community that we did not capture. This leads to a less than optimal compression.
- We can improve the compression of our sequence by using a hierarchical coding scheme that assigns nodes to four communities, where each community is encoded by two bits. We now only need two bits to identify one of the four nodes within each of the four communities.
- With this coding, we can store our sequence even more efficiently. For the trajectory above we only have four switches between communities, i.e. we need to store a total of $4 \times 2 = 8$ bits for community prefixes. We additionally have 32 transitions between nodes, each of which requires us to store two bits. This sums up to 72 bits or 2.25 bits per step. We have further compressed our sequence by using a community mapping that better matches to the “natural” community structure.
- We note that we have made a simplifying assumption: In the sequences above, we have included a delimiter (vertical bar) between the codewords for communities and nodes.In principle, we would need to account for this as well in the compression, however in the practice session and the exercise we will see that this can be avoided by a two-level prefix-free coding scheme.

Flow compression: overfitting

- consider mapping C_8 of nodes to **eight communities** $0, 1, \ldots, 7$
- use **hierarchical coding scheme** where nodes are identified relative to their community
  - 3 bits prefix identifies community
  - 1 bit suffix identifies node
- overfitting **decreases compression of flow**

example

sequence of 32 traversed nodes can be encoded with 2.5 bits per random walk step



000 0|1 001 0|1 000 0|1 ... 011 0|1 010 1|0|... 100 1|...

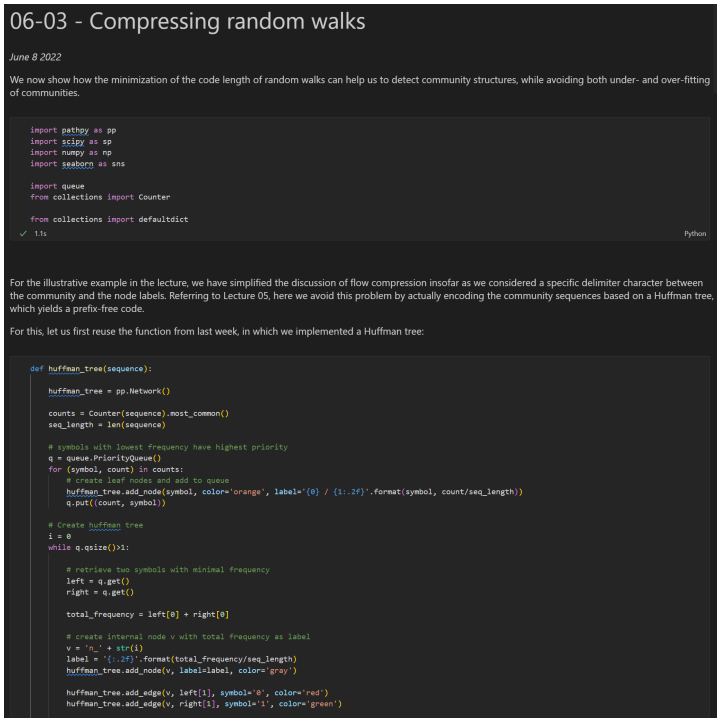
$$\underbrace{16 \cdot 3 \text{ bit}}_{\text{switch between clusters}} + \underbrace{32 \cdot 1 \text{ bit}}_{\text{switch between nodes}} = 80 \text{ bits}$$

Notes:

- What happens if we “overfit” the community structure in the graph, i.e. we use community labels that do not correspond to the “natural” four communities. We can, for instance, use eight communities, each of which is identified by three bits. In this case, a single bit is sufficient to uniquely identify the two nodes in each community.
- For the random walk sequence above, we now have 16 transitions between communities, each requiring three bits. For the additional 32 transitions between nodes, we now only need 32 bits in total. We obtain a total code length of 80 bits, which exceeds the 72 bits that we have found before.
- This has an intuitive interpretation: By overfitting the community structure we have saved an additional 32 bits of information needed to store the transitions between nodes, i.e. our community mapping better “explains”/compresses transitions.
- However, this comes at the expense that we have made our model more complex and we thus need 40 more bits to explain the switches between communities (i.e. our model). In this way, flow compression accounts both for the explanatory power of a model as well as for the complexity of the model and it thus inherently accounts for Occam’s razor! This makes it a very elegant solution for community detection, which yields both the optimal community assignments and the optimal number of communities simultaneously.
- A trivial mapping of each node to its own community (i.e. the solution with likelihood one in the stochastic block model) yields **no compression** just like the mapping of all nodes to a single community (the solution with minimal model complexity). We thus elegantly avoid trivial solutions.

Practice Session

- we use **Huffman coding to compress random walks** in a graph with communities
- we compare the **code length of encodings** using different community labels



06-03 - Compressing random walks

June 8 2022

We now show how the minimization of the code length of random walks can help us to detect community structures, while avoiding both under- and over-fitting of communities.

```
import pathpy as pp
import scipy as sp
import numpy as np
import seaborn as sns

import queue
from collections import Counter

from collections import defaultdict
```

✓ 1.1s Python

For the illustrative example in the lecture, we have simplified the discussion of flow compression insofar as we considered a specific delimiter character between the community and the node labels. Referring to Lecture 05, here we avoid this problem by actually encoding the community sequences based on a Huffman tree, which yields a prefix-free code.

For this, let us first reuse the function from last week, in which we implemented a Huffman tree:

```
def huffman_tree(sequence):

    huffman_tree = pp.Network()

    counts = Counter(sequence).most_common()
    seq_length = len(sequence)

    # symbols with lowest frequency have highest priority
    q = queue.PriorityQueue()
    for (symbol, count) in counts:
        # create leaf nodes and add to queue
        huffman_tree.add_node(symbol, color='orange', label='{0} / {1:.2f}'.format(symbol, count/seq_length))
        q.put((count, symbol))

    # Create huffman tree
    i = 0
    while q.qsize()>1:

        # retrieve two symbols with minimal frequency
        left = q.get()
        right = q.get()

        total_frequency = left[0] + right[0]

        # create internal node v with total frequency as label
        v = 'n_' + str(i)
        label = '{:.2f}'.format(total_frequency/seq_length)
        huffman_tree.add_node(v, label=label, color='gray')

        huffman_tree.add_edge(v, left[1], symbol='0', color='red')
        huffman_tree.add_edge(v, right[1], symbol='1', color='green')
```

practice session

see notebook 06-03 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

- In the third practice session, we use Huffman coding (see L05) to compress random walk sequences in a graph with four communities. Here we use an actual lossless coding scheme that does not require a delimiter between the community prefix and the node labels.

The MapEquation

- **asymptotic minimum per-symbol length of Huffman-coded random walk sequence** given by Shannon entropy H → Lecture 05
- for mapping $C_k : V \to \{1, \ldots, k\}$ of nodes to k communities, minimal code length $L(C_k)$ is given by the **MapEquation**

$$L(C_k) = \underbrace{qH(Q)}_{\substack{\text{switch between} \\ \text{communities}}} + \sum_{i=1}^{k} \underbrace{p_i H(P_i)}_{\substack{\text{switch between} \\ \text{nodes in community } i}}$$

- where $L(C_k)$ average number of bits per step and …
  - q is per-step probability that random walker switches between communities
  - Q are community visitation probabilities
  - p_i is per-step probability that random walker visits a node in community i
  - P_i are visitation probabilities of nodes in communities i

Notes:

- Rather than calculating the code length of an actual random walk sequence, we can apply Shannon's source coding theorem, which we introduced in the previous lecture. We can use this to directly calculate the minimal length of a lossless compression of a random walk in a graph. For this, we need the frequencies of symbols, which are given by the visitation probabilities of the random walk process.
- For a mapping C_k of nodes to k clusters, we use tbe Shannon entropy to calculate the minimal per-step expected code length of a random walk. For the simplest case of an undirected and unweighted graph, we obtain an expression that consists of two terms: The first term captures the minimal expected code length to encode transitions between communities. The second term captures the minimal expected code lengths to encode transitions between nodes in each of the k communities. The resulting **MapEquation** sums the entropies of the probability mass function of node and community visitations, weighted by relative probabilities of different transitions.
- That is, in addition to our illustrative example, we additionally account for the fact that –using Huffmann coding– we can encode random walks more efficiently if some nodes/communities are more frequently visited, e.g. we can choose shorter codes for more frequently visited nodes/communities.
- For a given mapping C_k and any number of communities k, summing these terms yields the minimal expected code length given a variable-length, prefix-free Huffman code to store sequences of nodes and cluster labels. We obtain a function that we can minimize across different community mappings and community numbers. Different from likelihood maximization in the stochastic block model, this includes the cost of model complexity in terms of the number of communities.

MapEquation: example

- consider clustering C_4 of nodes to **four clusters**

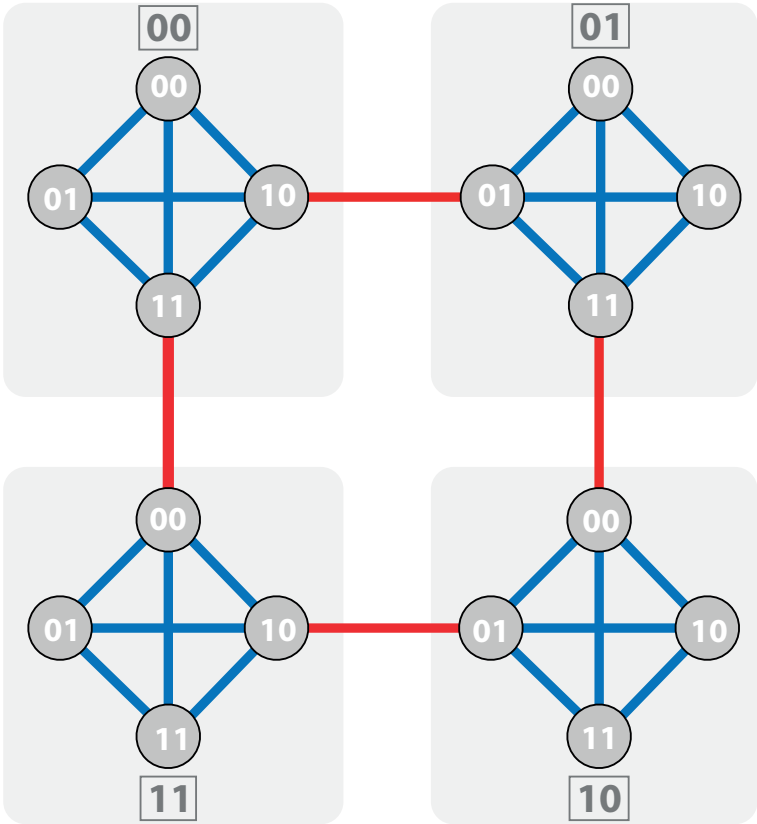
$$L(C_k) = \underbrace{qH(Q)}_{\text{switch between communities}} + \sum_{i=1}^{k} \underbrace{p_i H(P_i)}_{\text{switch between nodes in community } i}$$

- random walk traverses all edges E of **undirected network** with same probability (for $t \to \infty$) → exercise sheet

- for undirected graph $G = (V, E)$, **stationary visitation probabilities** are

$$\pi_v = \frac{d(v)}{2|E|} \text{for } v \in V$$

- makes it easy to calculate $L(C_k)$



$$L(C_4) \approx \frac{8}{56} \cdot 2 \text{ bit} + \sum_{i=1}^{4} \frac{14}{56} \cdot 1.99 \text{ bit} \approx 2.27 \text{ bit}$$

Notes:

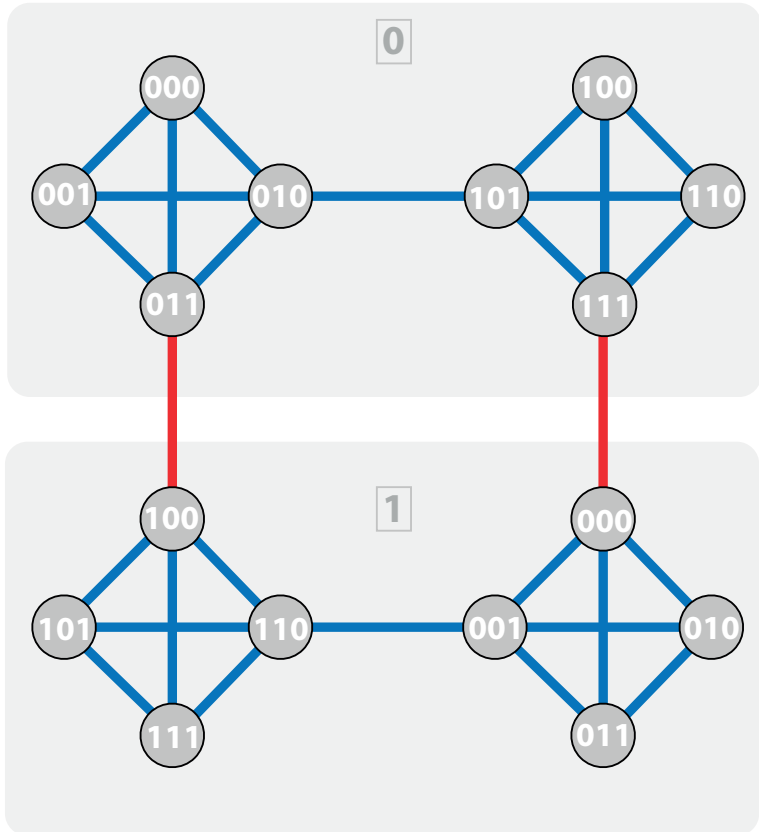
- We now apply this approach to our illustrative example. Rather than calculating the code length for one specific random walk trajectory, we can now calculate the expected minimal code length of a Huffman code, given a cluster mapping C. This raises the question how we can calculate the different quantities in the MapEquation that capture the probability of transition between nodes and clusters, as well as the visitation probabilities of nodes and clusters.
- For undirected and unweighted networks this is simple. For undirected/unweighted networks the stationary visitation probabilities of nodes after a long time $t \rightarrow$ is simply given by the node degrees divided by the sum of degrees. This is equivalent to saying that, in the stationary state, a random walk traverses each edge in the network with the same probability. This allows us to calculate q by counting the fraction of links that connect different clusters. For each p_i, we can simply sum the degrees of nodes in cluster i and divide it by the sum of all degrees (which is 56 in the example).
- Note that the entropy of each P_i is slightly below 2 because two of the nodes have a higher degree, and thus a higher visitation probabilities. This can be used to get a smaller expected code length, e.g. by means of Huffman coding.
- In summary, we can directly calculate the MapEquation based on the network topology. For the optimal mapping of nodes to clusters above, we obtain a minimal expected code length (per random walk step) of approx. 2.27 bits.

MapEquation: underfitting

- consider mapping C_2 of nodes two **two communities**

$$L(C_k) = \underbrace{qH(Q)}_{\text{switch between communities}} + \sum_{i=1}^{k} \underbrace{p_i H(P_i)}_{\text{switch between nodes in community } i}$$

- underfitting **increases minimal code length** $L(C_2)$



$$L(C_2) = \frac{4}{56} \cdot 1 \text{ bit} + \sum_{i=1}^{2} \frac{28}{56} \cdot 2.99 \text{ bits} \approx 3.06 \text{ bits}$$

Notes:

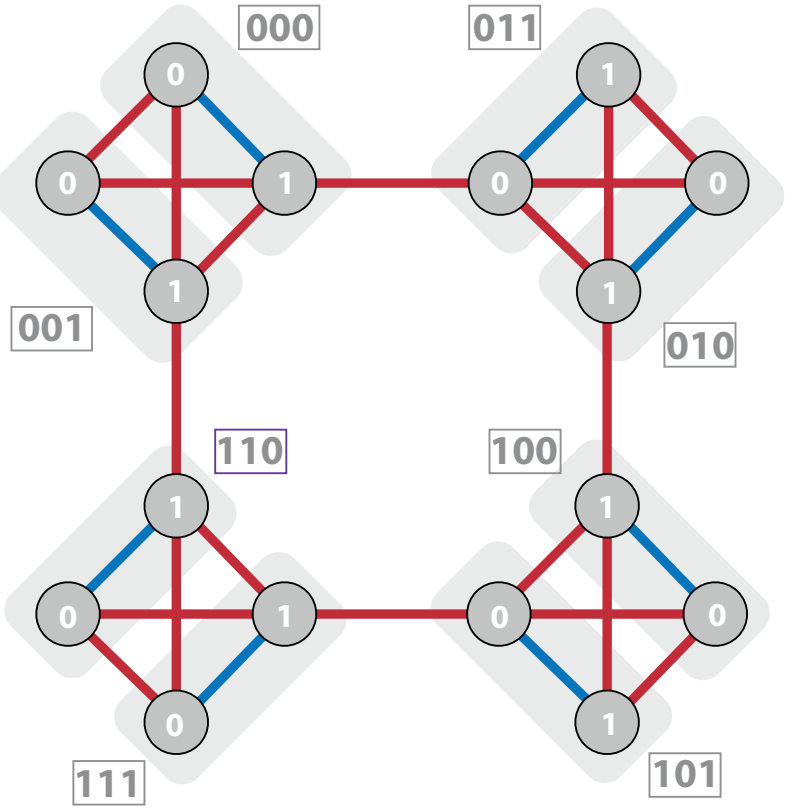
- Let us use the MapEquation to calculate the expected minimal code length for different community mappings, e.g. for the one that underfits the “true” community structure in the graph.
- In this case, the MapEquation reveals a code length of 3.06 bits, which shows that this community mapping is a less optimal model for the community pattern. We can find a mapping to more communities that would allow us to further compress the sequence.

MapEquation: overfitting

- consider mapping C_8 of nodes to **eight communities**

$$L(C_k) = \underbrace{qH(Q)}_{\text{switch between communities}} + \sum_{i=1}^{k} \underbrace{p_i H(P_i)}_{\text{switch between nodes in community } i}$$

- overfitting **increases minimum code length** $L(C_8)$



$$L(C_8) = \frac{40}{56} \cdot 3 \text{ bit} + \sum_{i=1}^{8} \frac{7}{56} \cdot 0.99 \text{ bit} \approx 3.13 \text{ bit}$$

Notes:

- We finally consider our example of a community mapping, where we assign the 16 nodes to eight communities. This overfits the community structure in the graph.
- Again, we find that the code length increases. This is due to the fact that the additional information needed to encode the sequence of community labels is not justified by the associated decrease of information needed to encode the transitions between nodes within those communities.

InfoMap algorithm

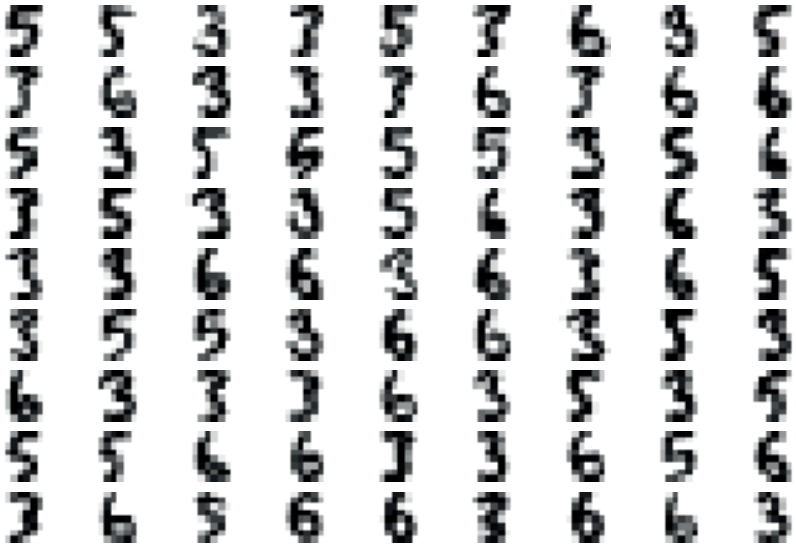
- MapEquation $L(C_k)$ calculates **minimal code length** of random walk for given community mapping C_k

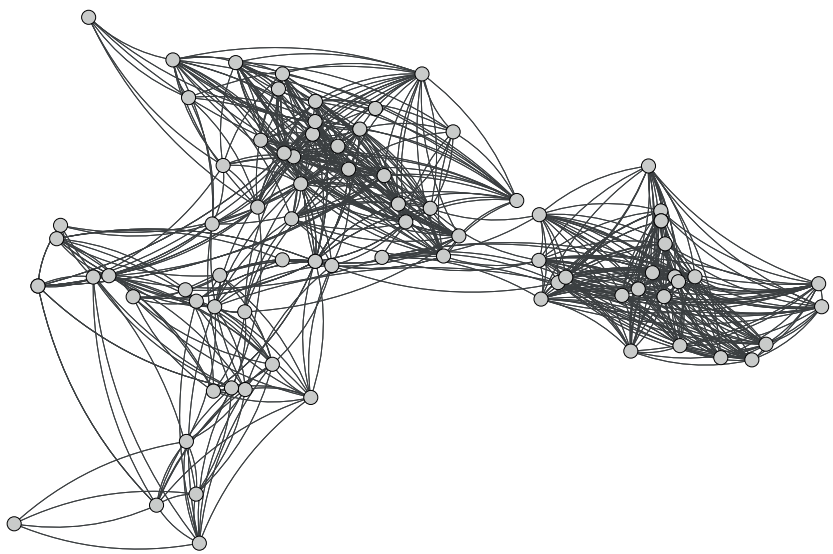
InfoMap

InfoMap algorithm uses MapEquation to find optimal cluster mapping $\hat{C}$ (with **any** number of communities) such that

$$\hat{C} = \underset{C}{\operatorname{argmin}}\, L(C)$$

→ M Rosvall, CT Bergstrom, 2008





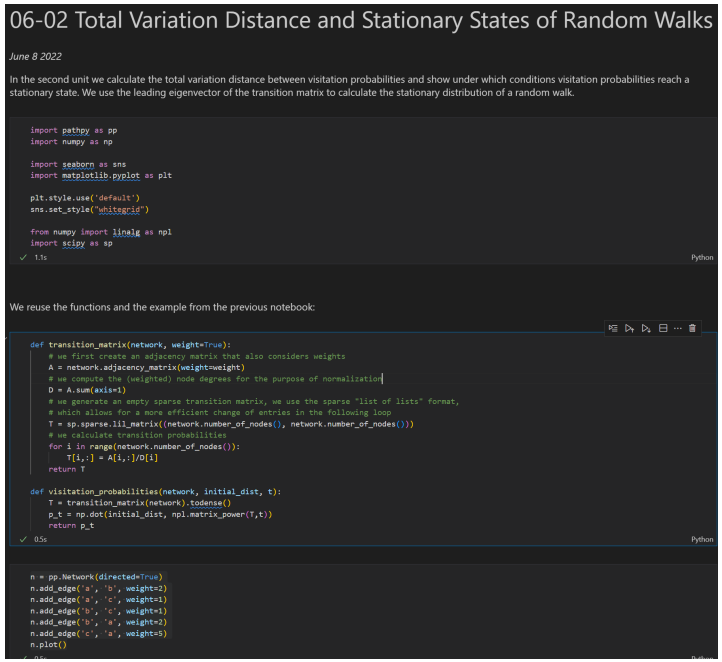
- we can use **heuristic optimization**, e.g., simulated annealing → L04 to find optimal communities $\hat{C}$

Notes:

- Given that we can compute the expected minimal code length for each given community mapping, how can we now find the **optimal community mapping**? Exhaustively enumerating community assignments and computing the MapEquation is not scalable in large graphs. We thus again need to apply heuristic optimization algorithms (e.g. variants of gradient ascent, simulated annealing, genetic algorithms, etc.). In the practice session, we will given an example for a simple greedy optimization algorithm.

Practice Session

- we implement the MapEquation for undirected networks
- we use simple greedy optimization to minimize the MapEquation
- we evaluate detected communities in an empirical network



06-02 Total Variation Distance and Stationary States of Random Walks

June 8 2022

In the second unit we calculate the total variation distance between visitation probabilities and show under which conditions visitation probabilities reach a stationary state. We use the leading eigenvector of the transition matrix to calculate the stationary distribution of a random walk.

```
import pathpy as pp
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

plt.style.use('default')
sns.set_style("whitegrid")

from numpy import linalg as npl
import scipy as sp
```

✓ 1.1s Python

We reuse the functions and the example from the previous notebook:

```
def transition_matrix(network, weight=True):
    # we first create an adjacency matrix that also considers weights
    A = network.adjacency_matrix(weight=weight)
    # we compute the (weighted) node degrees for the purpose of normalization
    D = A.sum(axis=1)
    # we generate an empty sparse transition matrix, we use the sparse "list of lists" format,
    # which allows for a more efficient change of entries in the following loop
    T = sp.sparse.lil_matrix((network.number_of_nodes(), network.number_of_nodes()))
    # we calculate transition probabilities
    for i in range(network.number_of_nodes()):
        T[i,:] = A[i,:]/D[i]
    return T

def visitation_probabilities(network, initial_dist, t):
    T = transition_matrix(network).todense()
    p_t = np.dot(initial_dist, npl.matrix_power(T,t))
    return p_t
```

✓ 0.5s Python

```
n = pp.Network(directed=True)
n.add_edge('a', 'b', weight=2)
n.add_edge('a', 'c', weight=1)
n.add_edge('b', 'c', weight=1)
n.add_edge('b', 'a', weight=2)
n.add_edge('c', 'a', weight=5)
n.plot()
```

practice session

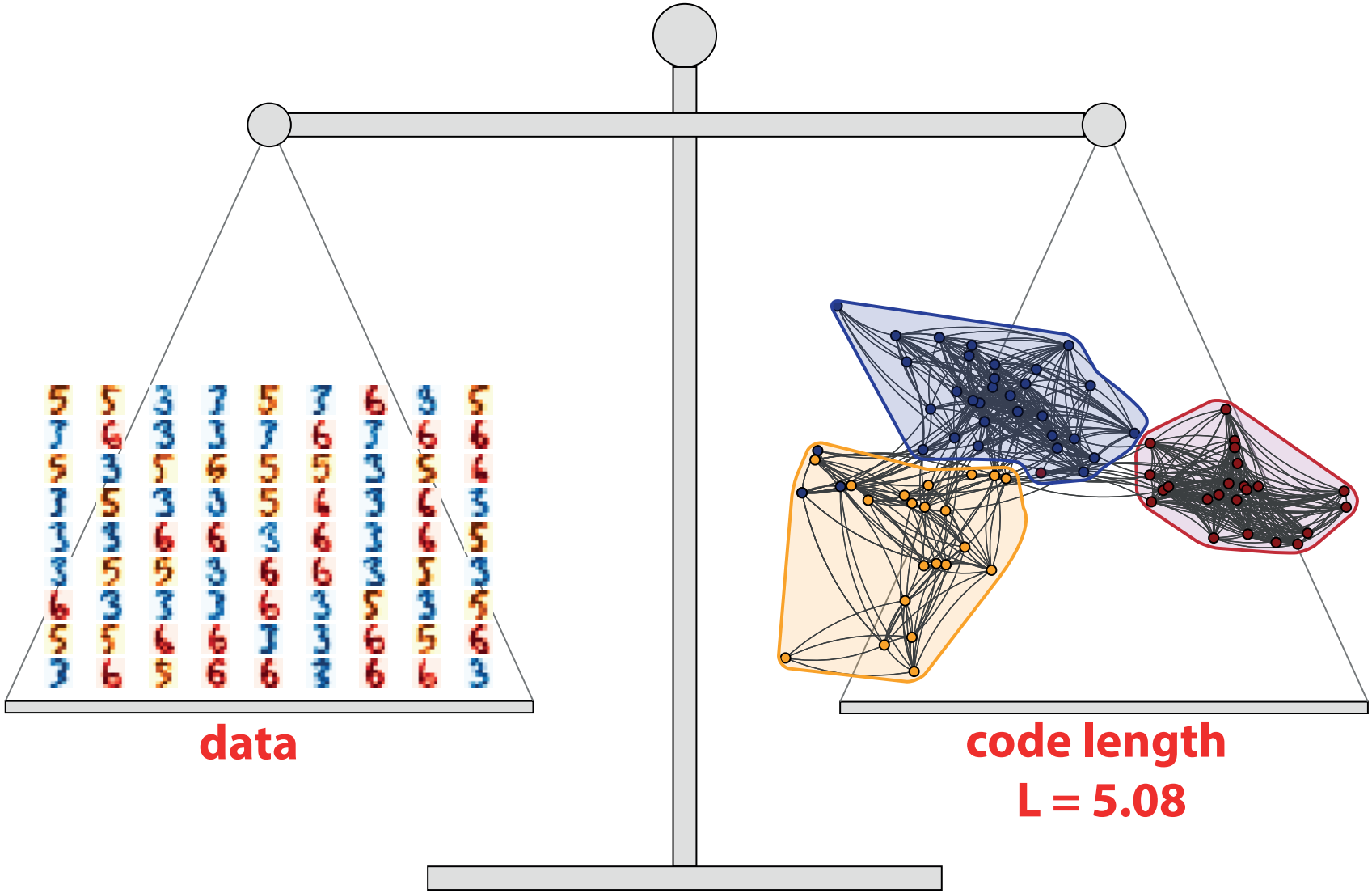
see notebook 06-04 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

- In the final practice session, we implement the MapEquation and use it to detect optimal community structures in undirected networks. We use a simple greedy optimization algorithm to find communities that minimize the MapEquation and apply the algorithm to the empirical Karate club network.

Model selection

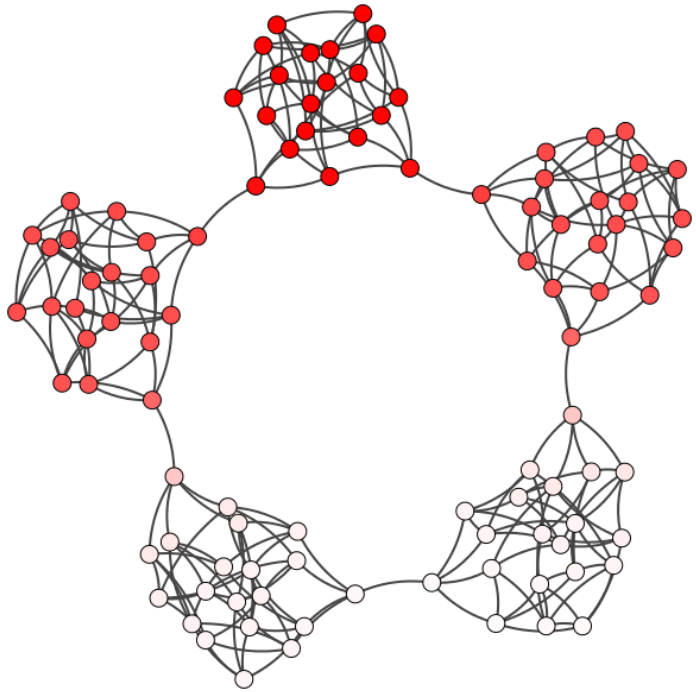


Notes:

- We now come back to the machine learning problem introduced last week. We have reformulated the problem of identifying clusters of similar hand-written digits as a community detection problem. For this, we first embedded pixel images of digits in a two-dimensional Euclidean space (by calculating the center of mass of pixel values). We then mapped this to a graph, where two nodes (representing images) are connected by an undirected edge if their distance is below a threshold ϵ
- We can now test whether InfoMap is able to find the optimal clustering of images. As an example, we compare three community assignments with two, three, and five clusters respectively. We then calculate the minimal description length of a random walk based on the MapEquation.
- We find that this data compression approach indeed identifies the optimal clustering of characters. The best model with two communities has a minimum (per-step) description length of $L(\vec{z}_2) = 5.66$ bits. The best model with five communities improves the description length to $L(\vec{z}_5) = 5.34$. The optimal model with a minimum description length of $L(\vec{z}_3) = 5.08$ bits has three communities, which almost perfectly correspond to the different digits in our data set.

In summary

- **random walks** are important foundation for network analysis and machine learning
- sequence of nodes visited by random walk captures **topological patterns in a graph**
- we can use **MapEuation** to compress random walks based on community structures
- highlights fundamental relation between **machine learning, model selection, and data compression**



Notes:

- In summary, we have introduced a model for random walks in graphs, which can be used to generate random sequences of nodes traversed in a network. This modelling framework is of crucial importance for network analysis and graph learning.
- In today's lecture, we have seen that we detect communities by rephrasing community detection as optimal compression of random walk sequences. We obtain an elegant algorithm that –apart from being a popular method to detect communities in large networks– highlights fundamental relations between unsupervised machine learning, model selection and data compression.

Questions

1. How is the transition matrix of a random walk defined for weighted and unweighted networks?
2. When does a random walk in a network converge to a unique stationary distribution?
3. How can we compute the stationary distribution of a random walk?
4. How is total variation distance defined? What is its maximum value?
5. For a right-stochastic transition matrix $\mathbf{T}$ and an initial distribution $\pi^{(0)}$, how can we compute visitation probabilities at time t of a random walk?
6. Can you give an example for a network in which a random walk never reaches a stationary distribution?
7. Explain the basic idea behind flow compression.
8. How is the MapEquation defined? How is it used in InfoMap?
9. How can we calculate the MapEquation for undirected and unweighted networks?
10. Explain how we can use PageRank to generalize InfoMap to directed, weighted networks that are weakly but not strongly connected.
11. How can we find a community mapping that minimizes the MapEquation?

Notes:

References

reading list

- AA Markov: **Extension of the limit theorems of probability theory to a sum of variables connected in a chain**, reprinted in Appendix B of *R. Howard: Dynamic Probabilistic Systems, volume 1: Markov Chains*, 1971
- O Perron: **Zur Theorie der Matrices**, Math. Ann. 64, 1907
- G Frobenius: **Über Matrizen aus nicht negativen Elementen**, Berl. Ber., 1912
- O Häggström: **Finite Markov Chains and Algorithmic Applications**, 2002
- DA Huffman: **A Method for the Construction of Minimum-Redundancy Codes**, Proceedings of the IRE, 1952
- M Rosvall and CT Bergstrom: **Maps of random walks on complex networks reveal community structure**, Proceedings of the National Academy of Sciences, 2008

PNAS

Maps of random walks on complex networks reveal community structure

Martin Rosvall*† and Carl T. Bergstrom*‡

*Department of Biology, University of Washington, Seattle, WA 98195-1800; and †Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM 87501

Edited by Brian Skyrms, University of California, Irvine, CA, and approved December 10, 2007 (received for review July 21, 2007)

To comprehend the multipartite organization of large-scale biological and social systems, we introduce an information theoretic approach that reveals community structure in weighted and directed networks. We use the probability flow of random walks on a network as a proxy for information flows in the real system and decompose the network into modules by compressing a description of the probability flow. The result is a map that both simplifies and highlights the regularities in the structure and their relationships. We illustrate the method by making a map of scientific communication as captured in the citation patterns of >6,000 journals. We discover a multicentric organization with fields that vary dramatically in size and degree of integration into the network of science. Along the backbone of the network—including physics, chemistry, molecular biology, and medicine—information flows bidirectionally, but the map reveals a directional pattern of citation from the applied fields to the basic sciences.

clustering | compression | information theory | map of science | bibiometrics

Biological and social systems are differentiated, multipartite, integrated, and dynamic. Data about these systems, now available on unprecedented scales, often are schematized as networks. Such abstractions are powerful (1, 2), but even as abstractions they remain highly complex. It therefore is helpful to decompose the myriad nodes and links into modules that represent the network (3–5). A cogent representation will retain the important information about the network and reflect the fact that interactions between the elements in complex systems are weighted, directional, interdependent, and conductive. Good representations both simplify and highlight the underlying structures and the relationships that they depict; they are maps (6, 7).

To create a good map, the cartographer must attain a fine balance between omitting important structures by oversimplification and obscuring significant relationships in a barrage of superfluous detail. The best maps convey a great deal of information but require minimal bandwidth: the best maps are also good compressions. By adopting an information-theoretic approach, we can measure how efficiently a map represents the underlying geography, and we can measure how much detail is lost in the process of simplification, which allows us to quantify and resolve the cartographer's tradeoff.

Network Maps and Coding Theory

In this article, we use maps to describe the dynamics across the links and nodes in directed, weighted networks that represent the local interactions among the subunits of a system. These local interactions induce a system-wide flow of information that characterizes the behavior of the full system (8–12). Consequently, if we want to understand how network structure relates to system behavior, we need to understand the flow of information on the network. We therefore identify the modules that compose the network by finding an efficiently coarse-grained description of how information flows on the network. A group of nodes among which information flows quickly and easily can be aggregated and described as a single well connected module; the links between modules capture the avenues of information flow between those modules.

Succinctly describing information flow is a coding or compression problem. The key idea in coding theory is that a data stream can be compressed by a code that exploits regularities in the process that generates the stream (13). We use a random walk as a proxy for the information flow, because a random walk uses all of the information in the network representation and nothing more. Thus, it provides a default mechanism for generating a dynamics from a network diagram alone (8).

Taking this approach, we develop an efficient code to describe a random walk on a network. We thereby show that finding community structure in networks is equivalent to solving a coding problem (14–16). We exemplify this method by making a map of science, based on how information flows among scientific journals by means of citations.

Describing a Path on a Network. To illustrate what coding has to do with map-making, consider the following communication game. Suppose that you and I both know the structure of a weighted, directed network. We aim to choose a code that will allow us to efficiently describe paths on the network that arise from a random walk process in a language that reflects the underlying structure of the network. How should we design our code?

If maximal compression were our only objective, we could encode the path at or near the entropy rate of the corresponding Markov process. Shannon showed that one can achieve this rate by assigning to each node a unique dictionary over the outgoing transitions (17). But compression is not our only objective; here, we want our language to reflect the network structure, we want the words we use to refer to things in the world. Shannon's approach does not do this for us because every codeword would have a different meaning depending on where it is used. Compare maps: useful maps assign unique names to important structures. Thus, we seek a way of describing or encoding the random walk in which important structures indeed retain unique names. Let us look at a concrete example. Fig. 1*A* shows a weighted network with $n = 25$ nodes. The link thickness indicates the relative probability that a random walk will traverse any particular link. Overlaid on the network is a specific 71-step realization of a random walk that we will use to illustrate our communication game. In Fig. 1, we describe this walk with increasing levels of compression (*B–D*), exploiting more and more of the regularities in the network.

Huffman Coding. A straightforward method of giving names to nodes is to use a Huffman code (18). Huffman codes save space

Author contributions: M.R. and C.T.B. designed research, performed research, and wrote the paper.

The authors declare no conflict of interest.

This article is a PNAS Direct Submission.

Freely available online through the PNAS open access option.

*To whom correspondence should be addressed. E-mail: rosvall@u.washington.edu.

This article contains supporting information online at www.pnas.org/cgi/content/full/0706851105/DC1.

© 2008 by The National Academy of Sciences of the USA

www.pnas.org/cgi/doi/10.1073/pnas.0706851105

Notes: