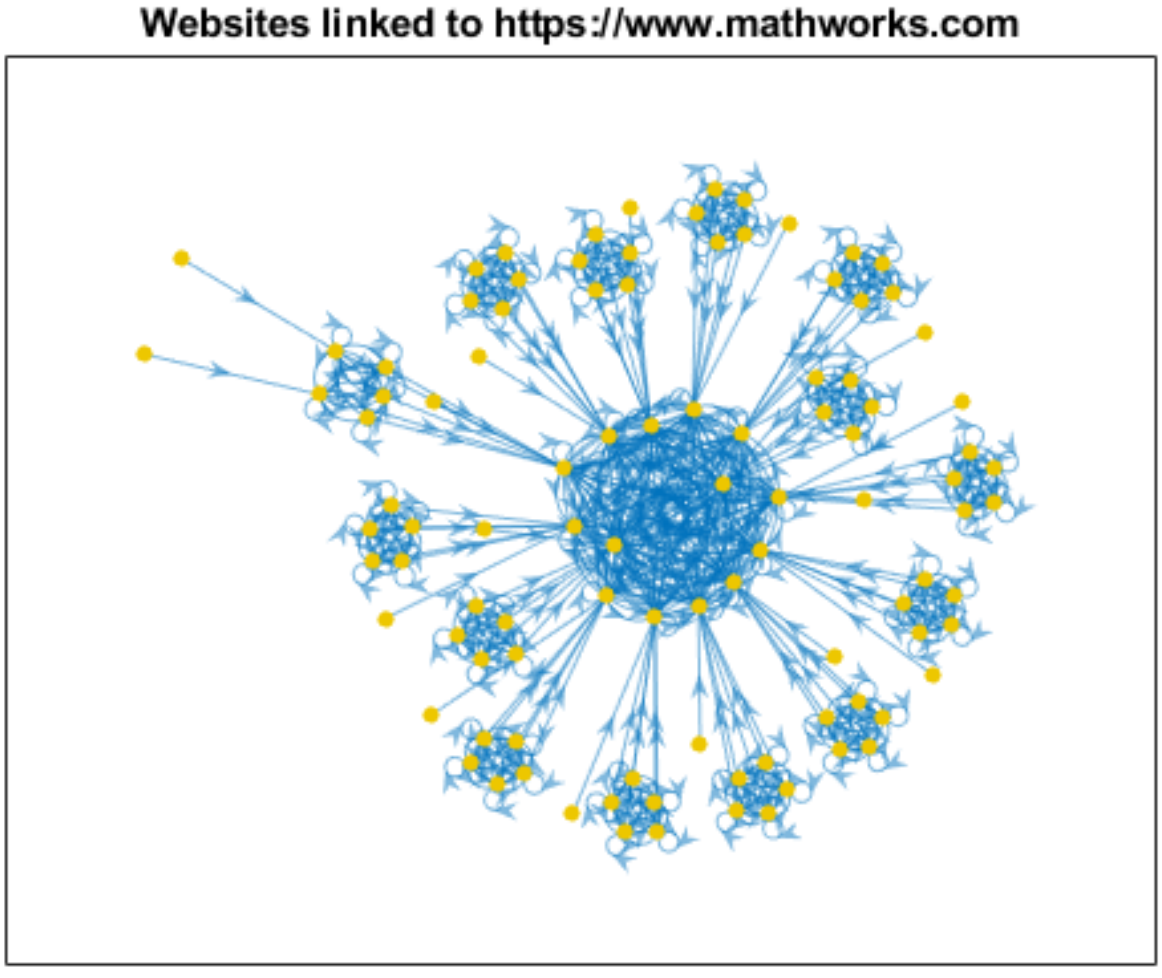
Math 260: Python programming in math

Fall 2020

Searching the internet:
PageRank and Markov chains

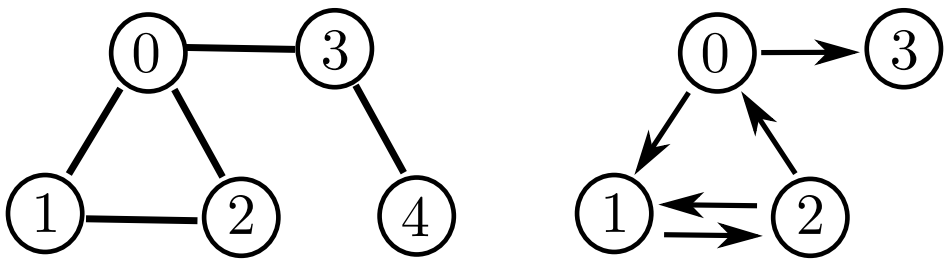
Graphs



Source: https://www.mathworks.com/help/matlab/math/use-page-rank-algorithm-to-rank-websites.html

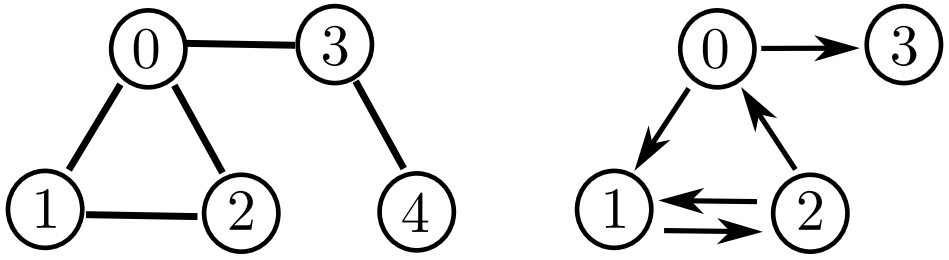
Graphs

- A **graph** is a set of **vertices** V connected by edges.
- The 'neighbors' of v are the vertices linked from v
- The 'edge set' E is the set of pairs (v, w) (edges from $v \in V$ to $w \in V$)
- a **directed graph** distinguishes between edges from $v \to w$ and $w \to v$ (e.g. links *from* web-pages)



Key question: how do we represent a graph in code?

Graphs



First, number the vertices $0, \cdots, N-1$. Then...

Option 1: Just list the edges...

- Create a list of edges (v_i, v_j) (tuples)
- Not very efficient!

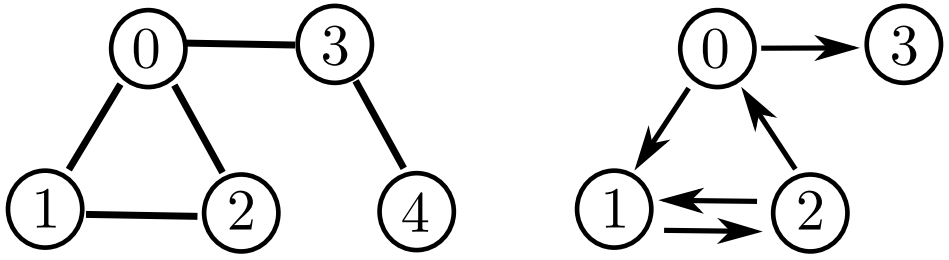
```
# directed example:
edges = [ [1,2],
          [2,1],
          [2,0],
          [0,1],
          [0,3]]
```

Option 2: Create an **adjacency list**

- Map k to neighbors of v_k
- Fast to look up all neighbors of v
- Just a list of 'lists of neighbors'

```
# directed example:
adj_list = [ [1,3],  # 0 -> 1, 3
             [2],  # 1 -> 2
             [0,1]
             []]
adj_list[0] # neighbors of vert. 0
```

Graphs



Another representation is the **adjacency matrix** A:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ links to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Adjacency matrices for the two graphs above:

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

e.g. first row of $A_1 \implies v_0$ linked to $1, 2, 3$ (not 0 or 4)

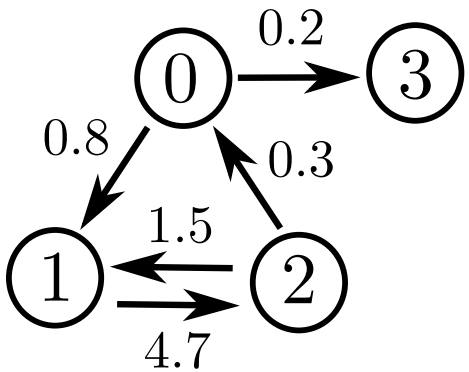
- Important matrix in graph theory!
- Typically sparse (mostly zeros)
- The adjacency list gives the non-zero entries in each row of A

Weighted graphs

A **weighted graph** associates a number w_{ij} to each edge $v_i \to v_j$.

- e.g cities connected by roads, weight = length of road
- We can keep track of this along with the adjacency matrix/list

A **weighted adjacency matrix** has the weight w_{ij} in the (i, j) entry, e.g.



$$A = \begin{bmatrix} 0 & 0.8 & 0 & 0.2 \\ 0 & 0 & 4.7 & 0 \\ 1.5 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

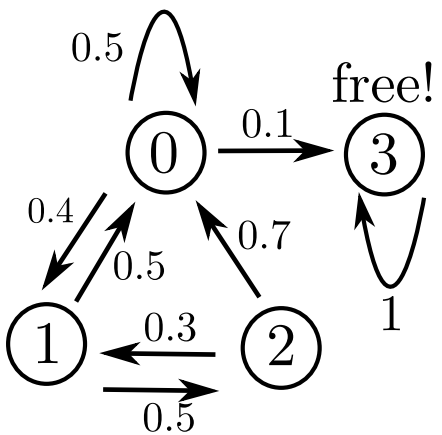
Now onto the object of interest here...

Discrete Markov chains

Consider a bee flying through a house with three rooms.

- Label the rooms $0, 1, 2$; you also open a window to the outside (room 3).
- The bee moves from one room to another at random each minute
- It chooses to go from room $r_i \to r_j$ with probability p_{ij}.

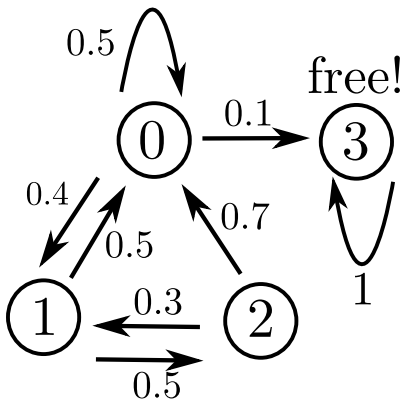
We can represent this process with a weighted directed graph, e.g.



$$P = \begin{bmatrix} 0.5 & 0.4 & 0 & 0.1 \\ 0 & 0.5 & 0.5 & 0 \\ 0.7 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The matrix P is the **transition matrix** - it describes the probability of transitioning from one place to the next.
- A process of this type is called a **(discrete) Markov chain**.

Discrete Markov chains



$$P = \begin{bmatrix} 0.5 & 0.4 & 0 & 0.1 \\ 0 & 0.5 & 0.5 & 0 \\ 0.7 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

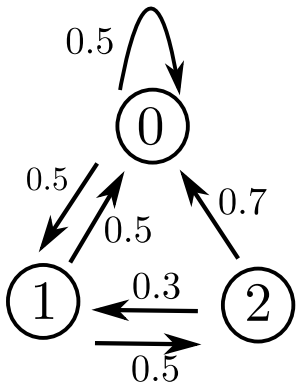
- It's easy to simulate directly given the adjacency list
- Suppose we have the adjacency list `adj` and a probability list `probs` so that, e.g. `adj[0]` $\rightarrow$ `[0,1,3]` and `probs[0]` $\rightarrow$ `[0.5, 0.4, 0.1]`)

```
while pos != 3:  # (while not free)
    r = random.uniform(0, 1)
    p = probs[pos]  # get list of transition probs.
    k = 0
    total = p[0]
    while r > total:
        k += 1
        total += p[k]
    pos = adj[pos][k]  # go to selected neighbor
```

Generate $x \in (0, 1)$, check if $x > p[0]$, then $p[0] < x < p[0] + p[1]$ and so on.

Discrete Markov chains

For more on the 'escape time' for the bee, see a probability course...
Let's now consider a variant. Suppose the window is closed...



$$P = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0.7 & 0.3 & 0 \end{bmatrix}$$

This Markov chain is called **recurrent** - the bee will wander in the chain forever.

- The key question: After a long time, what is the probability s_j that the bee will be in room j?
- This is independent of the bee's starting position
- Call this the 'stationary distribution' **s**.

Discrete Markov chains

A nice probability argument gives us a formula. Let

$$s_j = \text{probability the bee is in room } j \text{ after a long time}$$

and let p_{ij} be the transition probability for $i \to j$.

- If the bee is in room j, it must have come from room i, so

$$\text{prob(bee in room } j) = \sum_i \text{prob(bee in room } i \text{ the step before)} \cdot p_{ij}$$

- But after a long time, the LHS is just s_j and the prob in the sum is s_i, so

$$s_j = \sum_i s_i p_{ij}.$$

In matrix form, the result is that

$$\mathbf{s} = P^T \mathbf{s}.$$

That is, $\mathbf{s}$ is an eigenvector of P^T with eigenvalue 1.

Discrete Markov chains

To summarize the main result...

- Let P be the transition matrix for a recurrent Markov chain
  - p_{ij} is the probability to transition from $i \to j$
  - There are no 'dead ends': any vertex can be reached from any other
- Let s_j be the probability to be in vertex j after a long time (as a vector: $\mathbf{s}$)

Then the vector $\mathbf{s}$ (the stationary distribution) is given by

$$P^T \mathbf{s} = \mathbf{s}$$

i.e. an e-vector of P^T with e-value 1. Often we write $\mathbf{s}$ as a row vector, so

$$\mathbf{s} = \mathbf{s}P.$$

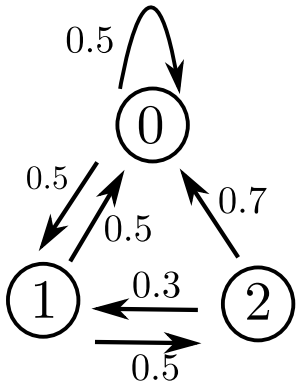
Theorem

A special case of the **Perron-Frobenius theorem** says that

- $\lambda = 1$ is the largest eigenvalue of P^T
- The eigenvector is unique (if scaled so that $\sum = 1$)

To find $\mathbf{s}$, we **solve an eigenvalue problem** for the largest eigenvalue of P^T.

Discrete Markov chains



$$P = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0.5 & 0.0 & 0.5 \\ 0.7 & 0.3 & 0 \end{bmatrix}$$

$$P^T = \begin{bmatrix} 0.5 & 0.5 & 0.7 \\ 0.5 & 0.0 & 0.3 \\ 0.0 & 0.5 & 0 \end{bmatrix}$$

Eigenvalues of P^T are 1 and $(-5 \pm i\sqrt{15})/20$ (not needed)

The eigenvector $\lambda = 1$ is $\mathbf{s} = (0.531, 0.31, 0.16)$ (scale so $\sum = 1$)

$$P^T \mathbf{s} = \mathbf{s}$$

$$\begin{bmatrix} 0.5 & 0.5 & 0.7 \\ 0.5 & 0.0 & 0.3 \\ 0.0 & 0.5 & 0 \end{bmatrix} \begin{bmatrix} 0.531 \\ 0.31 \\ 0.16 \end{bmatrix} = \begin{bmatrix} 0.531 \\ 0.31 \\ 0.16 \end{bmatrix}$$

so if you want to avoid the bee, you should be in room 2.

We really need a better way to find the eigenvector!

The power method

The goal is to find the **largest eigenvalue** (in magnitude) of an $n \times n$ matrix A.

First, an example. Consider

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}, \qquad \lambda_1 = 3,\ \mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \lambda_2 = 2,\ \mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Suppose we pick a starting vector, say, $\mathbf{x} = (6, -1)$. In the eigenvector basis,

$$\mathbf{x} = 5\mathbf{v}_1 + \mathbf{v}_2.$$

Now we repeatedly multiply by A on the left...

$$\begin{aligned} A\mathbf{x} &= 5 \cdot (3\mathbf{v}_1) + (2 \cdot \mathbf{v}_2) \\ A^2\mathbf{x} &= 5 \cdot (3^2\mathbf{v}_1) + (2^2\mathbf{v}_2) \\ A^k\mathbf{x} &= 5 \cdot 3^k\mathbf{v}_1 + 2^k\mathbf{v}_2 \quad \text{for } k \geq 1. \end{aligned}$$

The first term grows fastest, so it follows that

$$A^k\mathbf{x} \sim 5 \cdot 3^k\mathbf{v}_1 + (\text{smaller})$$

i.e. applying A repeatedly makes all but the $\mathbf{v}_1$ term smaller and smaller.

The power method

Now for the general case... For simplicity, assume that:

- A has n eigenvalues with $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$
- A has eigenvectors $\mathbf{v}_1, \cdots, \mathbf{v}_n$ that form a basis for $\mathbb{R}^n$

Pick any vector $\mathbf{x}_0$ and consider the simple iteration

$$\mathbf{x}_k = A\mathbf{x}_{k-1}$$

Then

$$\mathbf{x}_k = A^k\mathbf{x}_0 = c_1\lambda_1^k\mathbf{v}_1 + (\text{smaller terms}).$$

- We are free to rescale $\mathbf{x}$ at each step so the first term stays the same size:

$$\mathbf{x}_k = \frac{A\mathbf{x}_{k-1}}{\|A\mathbf{x}_{k-1}\|}$$

  where $\|w\|$ is the magnitude of a vector: $\|w\| = \sqrt{w_1^2 + \cdots + w_n^2}$.
- After doing so, the result is

$$\mathbf{x}_k \sim c\mathbf{v}_1 + (\text{terms that go to zero}).$$

  where c is such that $\|c\mathbf{v}_1\| = 1$.
- To get the eigenvalue, see the next slide...

The power method: finding eigenvalues (aside)

How do we get the eigenvalue? We want a ratio like

$$\frac{\mathbf{x}_{k+1}}{\mathbf{x}_k} \sim \lambda_1$$

but that doesn't work since $\mathbf{x}_k$ is a vector. Instead:

- Pick a vector $\mathbf{w}$
- Take a dot product with $\mathbf{w}$ to get a scalar
- Take the ratio of these dot products:

$$r_k = \frac{\mathbf{w} \cdot \mathbf{x}_{k+1}}{\mathbf{w} \cdot \mathbf{x}_k}$$

From the expression

$$\mathbf{x}_k \sim c_1 \lambda_1^k \mathbf{v}_1$$

we can show that the ratios r_k approach λ_1.
A better choice (effectively $\mathbf{w} = \mathbf{x}_k$) is the **Rayleigh quotient**

$$r_k = \frac{\mathbf{x}_k \cdot (A\mathbf{x}_k)}{\mathbf{x}_k \cdot \mathbf{x}_k} \to \lambda_1 \text{ as } k \to \infty.$$

Since we chose $\mathbf{x}_k$ to be a unit vector, the denominator vanishes, leaving

$$r_k = \mathbf{x}_k \cdot (A\mathbf{x}_k).$$

The power method

To summarize: suppose A is $n \times n$ with a largest eigenvalue λ_1 in magnitude. To find it and the eigenvector,

- Pick a random starting vector $\mathbf{x}_0$
- Compute (with $\|\mathbf{w}\| = \sqrt{w_1^2 + \cdots + w_n^2}$.) the iteration

$$\mathbf{x}_k = A\mathbf{x}_{k-1}/\|A\mathbf{x}_{k-1}\|, \qquad r_k = \mathbf{x}_k \cdot (A\mathbf{x}_k)$$

Then $r_k \to \lambda_1$ and $\mathbf{x}_k$ converges to an eigenvector of λ_1.

A simple python 'sketch':

```
def power_method(a):
    n = a.nrows  # number of rows
    x = # (set to random vector)
    while condition:
        q = multiply(A, x)
        r = dot_prod(x, q) #x^TAx
        x = q/norm(q) # normalize
    return x, r

def norm(x):
    return sqrt(sum((v**2 for v in x)))
```

Using numpy (sketch):

```
def power_method(a):
    n = a.shape[0]
    x = #...set to random np.array
    while condition:
        q = np.dot(a, x)
        r = x.dot(q)
        x = q/sqrt(q.dot(q))
    return x, r
```

The 'condition' needs to be specified here: One can stop when the r is close enough to converged (e.g. when $|r_k - r_{k-1}|$ is small enough).

For Markov chains

Now back to Markov chains...

- We want to find the eigenvector for $\lambda_1 = 1$ of the matrix $A = P^T$
- λ_1 is known to be the largest

The fact that $\lambda_1 = 1$ makes life easier! We have

$$A^k \mathbf{x}_0 \sim c_1 \mathbf{v}_1 + \cdots \implies A^k \mathbf{x}_0 \sim c_1 \lambda_1 + (\text{small}).$$

To be safe, it may be important to ensure the 'eigenvector' is a vector of probabilities, i.e.

$$\|\mathbf{x}_k\|_1 = 1 \text{ where } \|\mathbf{w}\|_1 := \sum_{j=1}^{n} |w_j|.$$

This is satisfied if $\|\mathbf{x}_0\|_1 = 1$, but rounding error may cause small deviations. Example code:

```
def stationary(p, steps):
    n = a.shape[0]
    x = #...set to random np.array, pos. values
    x = x/sum(x) # normalize
    for it in range(steps)
        x = np.dot(a, x)
        x = x/sum(x)  # to be safe...
    return x
```

PageRank

Finally, we can apply this idea to a search engine.

Consider a set of N web pages with (directed) links.

- Let A be the adjacency matrix for this directed graph
- Not all nodes have to be reachable from all others!

We define a Markov process by imagining a 'web surfer':

- At each step, the surfer picks a link at random, all with equal probability
- Let ℓ_i denote the number of outgoing links from page i

The transition probabilities are then

$$p_{ij} = \frac{a_{ij}}{\ell_i}$$

and we must solve

$$P^T \mathbf{s} = \mathbf{s}.$$

This would be plugged into the power method.

But there's a catch! The surfer needs somewhere to go if they get 'stuck' in a part of the graph with no links back.

PageRank

A simple fix - add artificial links to all pages.

- The surfer goes to a random page with probability $1 - \alpha$
- Take α to be near one (but not equal to one)

Then the modified transition probability is

$$\tilde{p}_{ij} = \alpha \frac{a_{ij}}{\ell_i} + \frac{1-\alpha}{N}.$$

Thus we must solve

$$M\mathbf{s} = \mathbf{s}$$

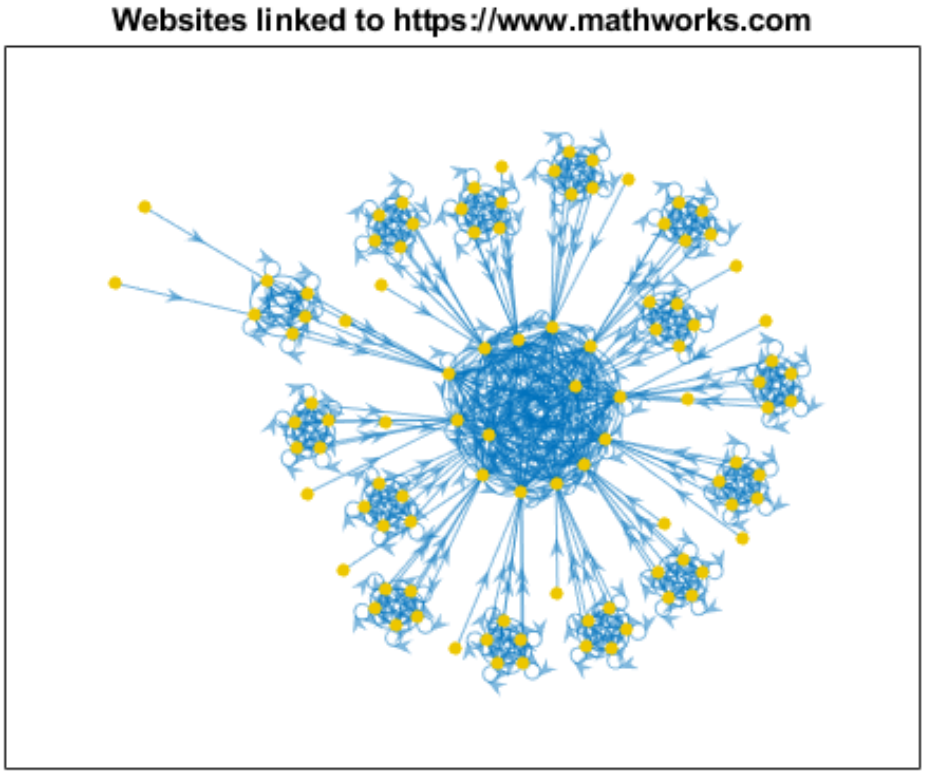
for the eigenvector $\mathbf{s}$, where M is the 'PageRank' matrix

$$M = \alpha P^T + \frac{(1-\alpha)}{N} E, \qquad E = \text{matrix of all ones.}$$

Note that the adjacency list `adj` for A, ℓ_i is just the size of `adj[i]`.

PageRank

Let's return to the `mathworks.com` example...

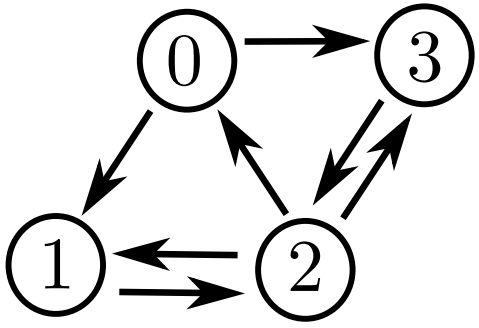


This data set has 100 pages, stored in a `.txt` file as an adjacency list, e.g.

```
...
14, 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15, 1
16, 1 16 17 18 19 20
...
```

PageRank

A small example:

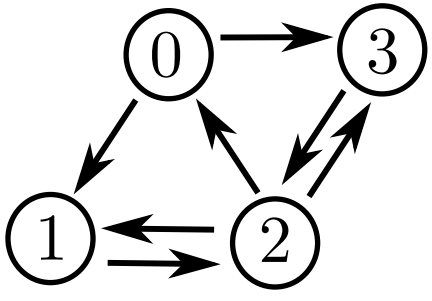


$$P = \begin{bmatrix} 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 \\ 1/3 & 1/3 & 0 & 1/3 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$M = \alpha P^T + \frac{1-\alpha}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Applying the power method with M and $\alpha = 0.9$ we get

$$M\mathbf{s} = \mathbf{s}, \qquad \mathbf{s} \approx (0.15, 0.22, 0.42, 0.22)^T$$



This ranks the websites by a measure of how connected they are to the pages.

Page 2 is highest ranked!

Aside: what do the iterates mean?

An aside: given a transition matrix P, the power method calculates

$$\mathbf{x}_0, \quad \mathbf{x}_1 = P^T\mathbf{x}_0, \quad \mathbf{x}_2 = (P^T)^2\mathbf{x}_0, \cdots$$

for a starting distribution $\mathbf{x}_0$, and we have

$$\lim_{k\to\infty} (P^T)^k\mathbf{x}_0 = \mathbf{s} \quad \text{(stationary dist.)}$$

But what do the iterates mean? We have that

$$P(\text{in state } j \text{ at step } 1) = \sum_i P(\text{in state } i \text{ at step } 0)p_{ij}.$$

- Let $\mathbf{x}_k = (P^T)^k\mathbf{x}_0$ (the power method iterate)
- The formula says that

$$(\mathbf{x}_1)_j = (P^T\mathbf{x}_0)_j = P(\text{in state } j \text{ at step } 1)$$

- so $\mathbf{x}_1$ is the distribution at step 1 (given $\mathbf{x}_0$)
- ... and $\mathbf{x}_k$ is the distribution at step k (given $\mathbf{x}_0$)

Aside: what do the iterates mean?

Example (from before, with $\alpha = 1$):

$$P = \begin{bmatrix} 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 \\ 1/3 & 1/3 & 0 & 1/3 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{x}_0 = (0.25, 0.25, 0.25, 0.25)$$

After one step...

$$\mathbf{x}_1 = [0.083, 0.208, 0.5 \quad, 0.208]$$

and so on...

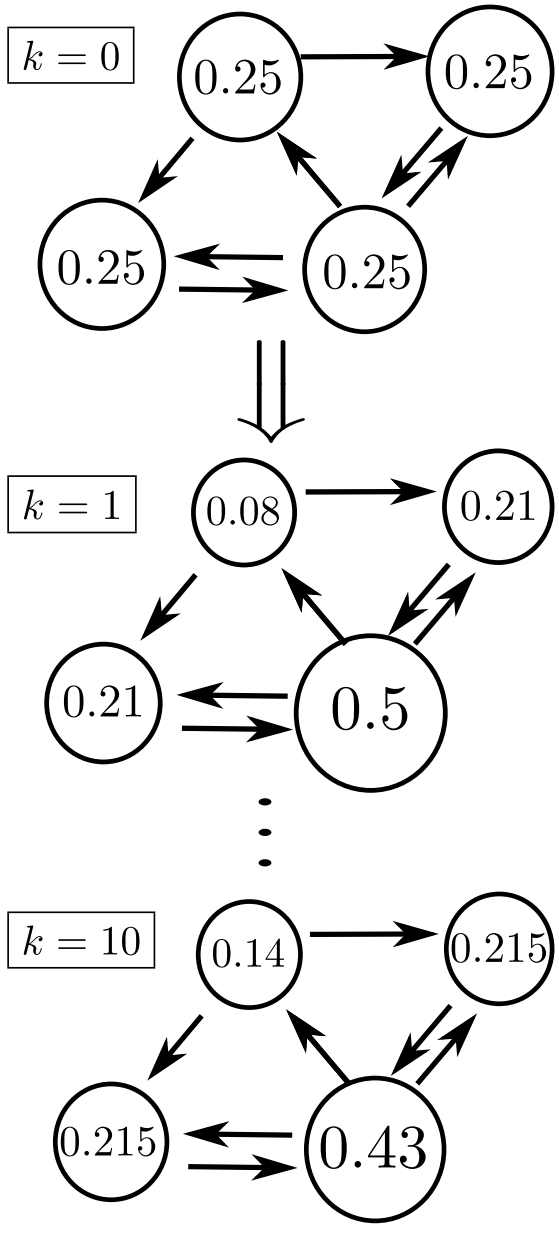
$$\mathbf{x}_2 = [0.167, 0.208, 0.417, 0.208]$$

$$\vdots = \quad \vdots$$

$$\mathbf{x}_9 = [0.143, 0.214, 0.429, 0.214]$$

$$\mathbf{x}_{10} = [0.143, 0.215, 0.428, 0.215]$$

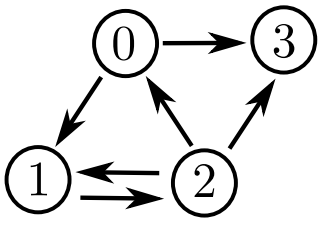
(converges to $\mathbf{s}$ as $k \to \infty$!)



PageRank

What about 'dead ends'?

- Cases matter for the model (should one-way links be important?)
- A few trategies exist.,, (not detailed here)
- Example: ($\alpha = 0.6$)



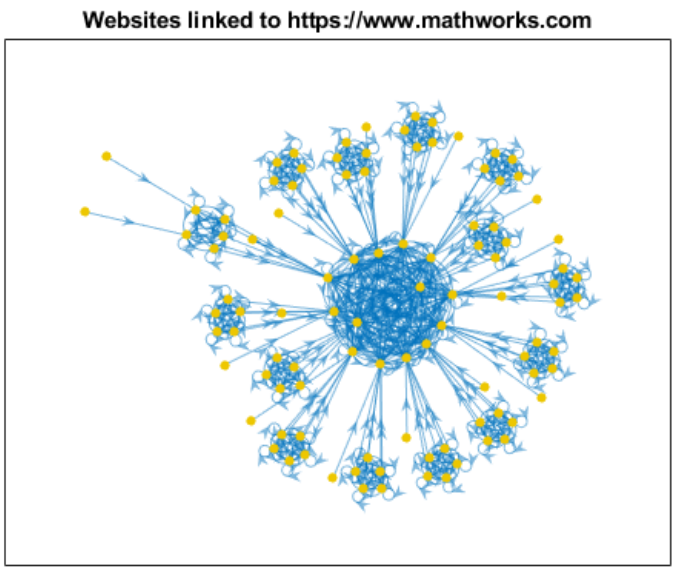
$$P = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad M = \begin{bmatrix} 0.1 & 0.1 & 0.3 & 0.1 \\ 0.4 & 0.1 & 0.3 & 0.1 \\ 0.1 & 0.7 & 0.1 & 0.1 \\ 0.4 & 0.1 & 0.3 & 0.1 \end{bmatrix}$$

The issue: M^T is no longer a transition matrix! A surfer that goes to the dead page disappears. The largest eigenvalue is less than one.

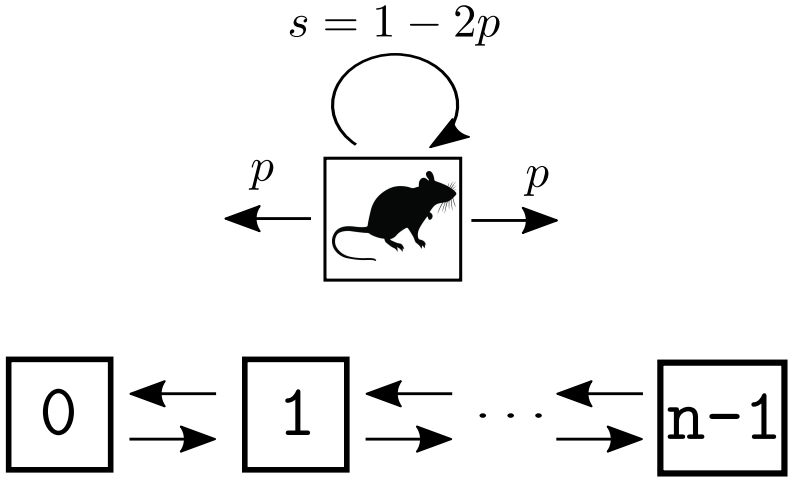
- Interpretation: $M^k\mathbf{x} \to 0$ as $k \to \infty$ - all surfers end up vanishing
- Good news: the power method still works!
- `x /= sum(x)` normalization is now required

Result for above: $\lambda_1 \approx 0.771$ and $\mathbf{s} \approx (0.161, 0.255, 0.330, 0.255)$

Sparse matrices



- Graph of $n = 100$ websites
- Each site: less than $k \approx 15$ links
- Adjacency matrix: $O(n^2)$ entries!



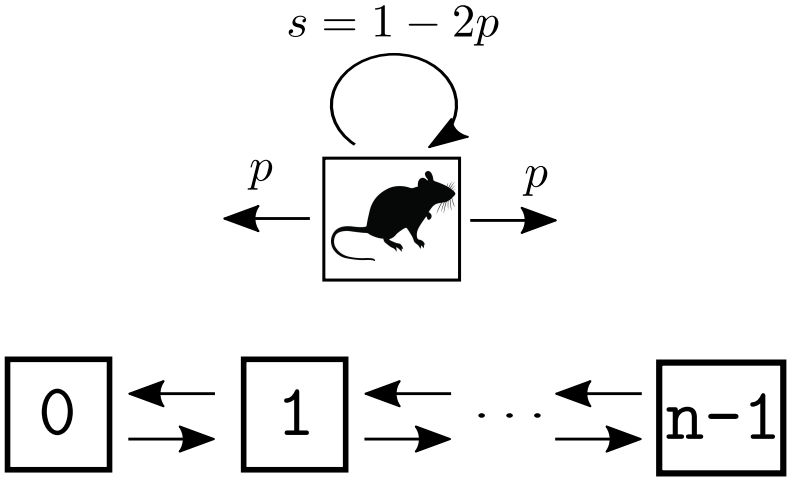
- Rat leaves a room with prob. p
- n total states
- Each state has $2 - 3$ neighbors

- Adjacency matrix has $O(N)$ non-zeros ($\ll N^2$)
- We must **store matrices in a compact form**!

Sparse matrices

Rat transition matrix ($s = 1 - 2p$):

$$\begin{bmatrix} s & p & 0 & \cdots & 0 \\ p & s & p & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & p & s & p \\ 0 & \cdots & 0 & p & s \end{bmatrix}$$



- The 'list of entries' and 'adjacency list' structures work here
- Adjacency list is more compact; list of entries is easier
- (For precise details, see 'compressed row/column format')

Sparse matrices

How to do this in python...? The 'general' way:

1) First, we need the 'list of entries' form:

$$\begin{bmatrix} s & p & 0 & \cdots & 0 \\ p & s & p & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & p & s & p \\ 0 & \cdots & 0 & p & s \end{bmatrix}$$

```
# initialize row, col, val
row[0:2] = 0
col[0:2] = [0, 1]
val[0:2] = [s, p]
pos = 2
for j in range(n-1):
    row[pos:pos+3] = [j, j, j]
    col[pos:pos+3] = range(pos, pos+3)
    val[pos:pos+3] = [p, s, p]
#... also last row ...
```

2) Next, use the sparse matrix class in `scipy.sparse`

```
from scipy import sparse

# given row, col, val, n
mat = sparse.coo_matrix((val, (row, col)), shape=(n, n))
```

Sparse matrices

What do we want to do with sparse matrices? Examples:

- Slices of rows/columns, submatrices etc.
- Matrix-vector products (important!) Ax
- Calculating eigenvalues
- Solving linear systems $Ax = b$

Sparse linear algebra deals with (good) algorithms to these.

`scipy.sparse` implements efficient methods for sparse matrices...

```
sparse_mat = sparse_coo(...)  # sparse matrix
dense_mat = sparse_mat.toarray()  # 2d array version
x = some_vector()

y = dense_mat.dot(x)   # regular multiply
y = sparse_mat.dot(x)  # uses sparse multiply
```

Sparse matrices

Important point:

The "COO" (list of entries) format is **not efficient** for most calculation!
The matrix should be converted to an efficient form before use.

- There are tradeoffs between efficiency and flexibility
- Formats include 'compressed column/row' (`scipy`: `csc` and `csr`)
- Conversion is typically fast between formats
  (but you can also construct each type directly)

Typical use: Construct COO (simple), then convert

```
#Conversion between types:
mat = sparse.coo_matrix(...)  # in `list of entries' form
mat.tocsr()  # convert to compressed row
mat.tocsc()  # convert to compress column

mat.toarray()  # convert to *dense* 2d array
```

Sparse matrices

Special case: banded matrix, e.g. tridiagonal:

$$A = \begin{bmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ c_2 & a_2 & b_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & c_n & a_n \end{bmatrix}$$

You can store this just as an $n \times 3$ array!

```
# sketch of data storage:
a = [[a1, b1, 0],
     [c2, a2, b2],
     ...
     [0, cn, an]]
```

(In general, k diagonals $\implies$ $n \times k$ array)