Linear programming

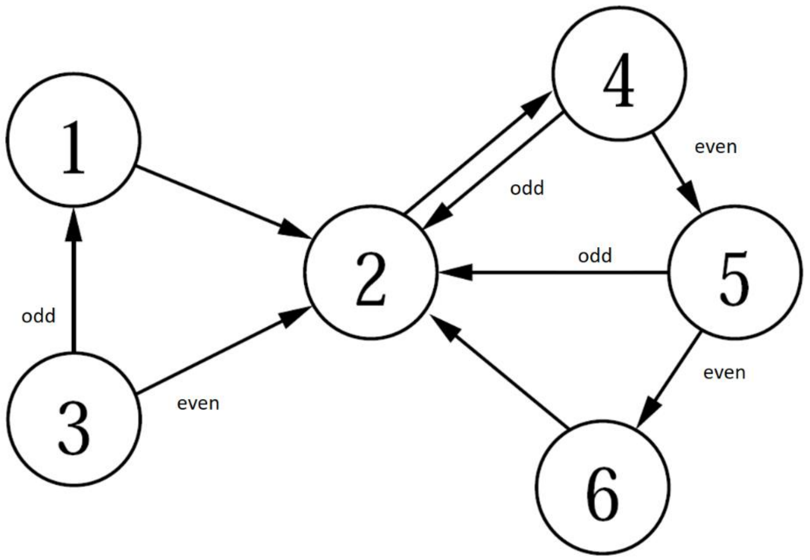
- Example Numpy: PageRank
- scipy.optimize.linprog
- Example linear programming: Maximum flow

PageRank

PageRank - A NumPy / Jupyter / matplotlib example

Central to Google's original search engine was the ranking of webpages using PageRank.

View the internet as a graph where nodes correspond to webpages and directed edges to links from one webpage to another webpage.



In the following we consider a very simple graph with six nodes and where every node has one or two outgoing edges.

The original description of the PageRank computation can be found in the research paper below containing an overview of the original infrastructure of the Google search engine.

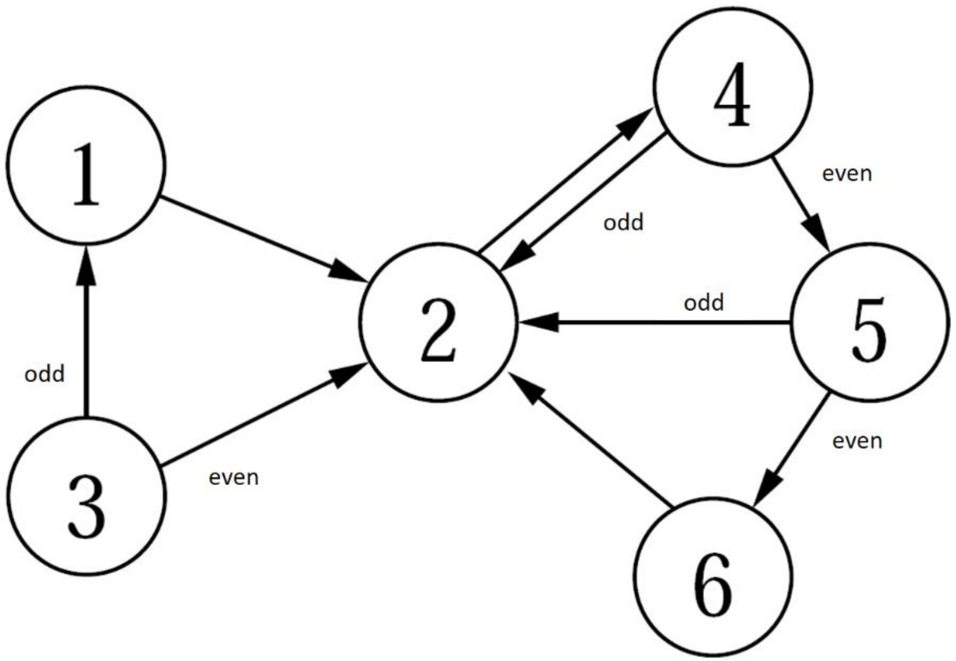
- Sergey Brin and Lawrence Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Seventh International World-Wide Web Conference (WWW 1998). [http://ilpubs.stanford.edu:8090/361/]

Five different ways to compute PageRank probabilities

1) Simulate random process manually by rolling dices
2) Simulate random process in Python
3) Computing probabilities using matrix multiplication
4) Repeated matrix squaring
5) Eigenvector for $\lambda = 1$

Random surfer model (simplified)

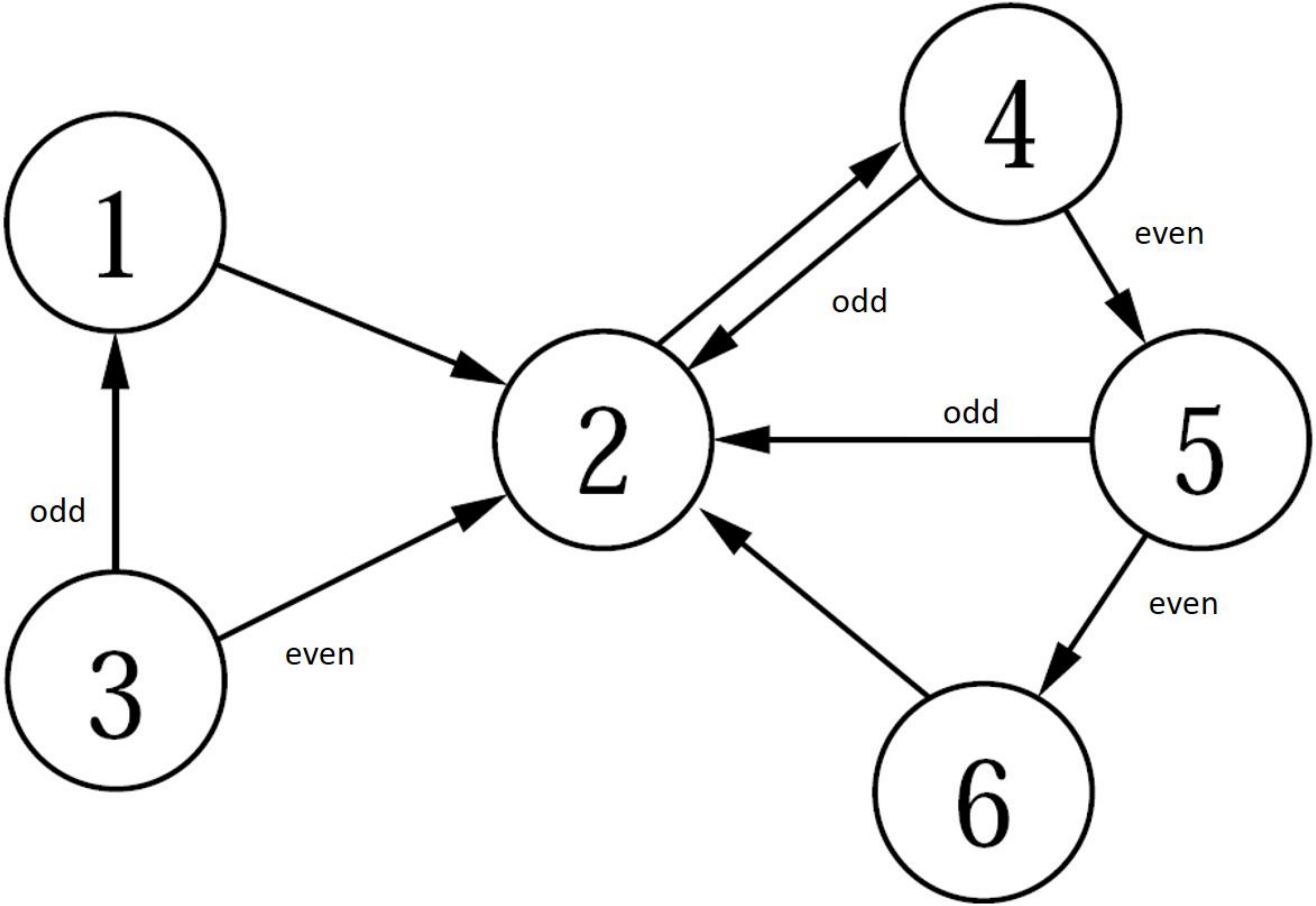
The PageRank of a node (web page) is the fraction of the time one visits a node by performing an *infinite random traversal* of the graph where one starts at node 1, and in each step performs:



- with probability 1/6 jumps to a random page (probability 1/6 for each node)
- with probability 5/6 follows an outgoing edge to an adjacent node (selected uniformly)

The above can be simulated by using a dice: Roll a *dice*. If it shows 6, jump to a random page by rolling the dice again to figure out which node to jump to. If the dice shows 1-5, follow an outgoing edge - if two outgoing edges roll the dice again and go to the lower number neighbor if it is odd.

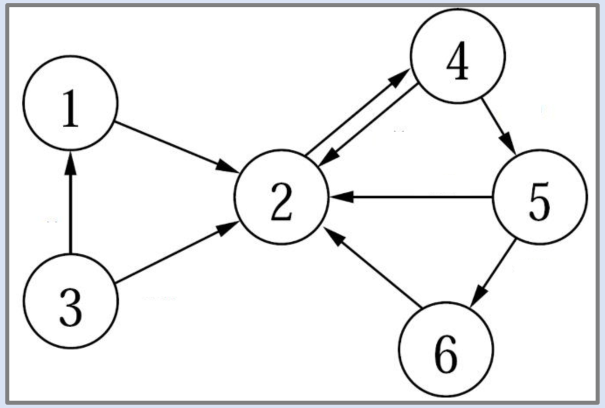
tiny.cc/pagerank


1
2
3
4
5
6
odd
even
odd
even
odd
even

Adjacency matrix and degree vector

pagerank.ipynb

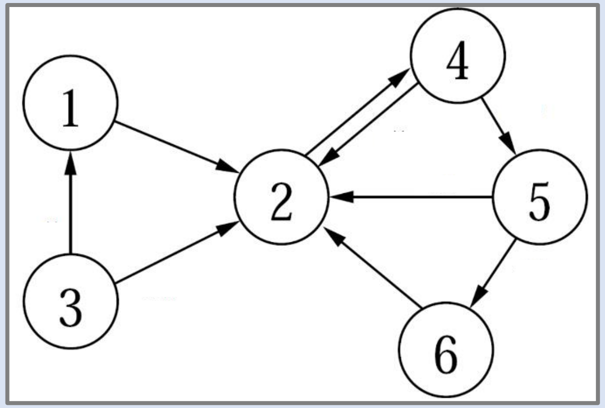
```
import numpy as np

# Adjacency matrix of the directed graph in the figure
# (note that the rows/colums are 0-indexed, whereas in the figure the nodes are 1-indexed)

G = np.array([[0, 1, 0, 0, 0, 0],
              [0, 0, 0, 1, 0, 0],
              [1, 1, 0, 0, 0, 0],
              [0, 1, 0, 0, 1, 0],
              [0, 1, 0, 0, 0, 1],
              [0, 1, 0, 0, 0, 0]])

n = G.shape[0]  # number of rows in G
degree = np.sum(G, axis=1, keepdims=1)  # creates a column vector with row sums = out-degrees

# The below code handles sinks, i.e. nodes with outdegree zero (no effect on the graph above)

G = G + (degree == 0)  # add edges from sinks to all nodes
degree = np.sum(G, axis=1, keepdims=1)
```



Simulate random walk (random surfer model)

pagerank.ipynb

```
from random import randint
STEPS = 1000000
# adjacency_list[i] is a list of all j where (i, j) is an edge of the graph.
adjacency_list = [[col for col in range(n) if G[row, col]] for row in range(n)]
count = [0] * n  # histogram over number of node visits
state = 0  # start at node with index 0
for _ in range(STEPS):
    count[state] += 1
    if randint(1, 6) == 6:  # original paper uses 15% instead of 1/6
        state = randint(0, 5)
    else:
        d = len(adjacency_list[state])
        state = adjacency_list[state][randint(0, d - 1)]
print(adjacency_list, [c / STEPS for c in count], sep="\n")
```



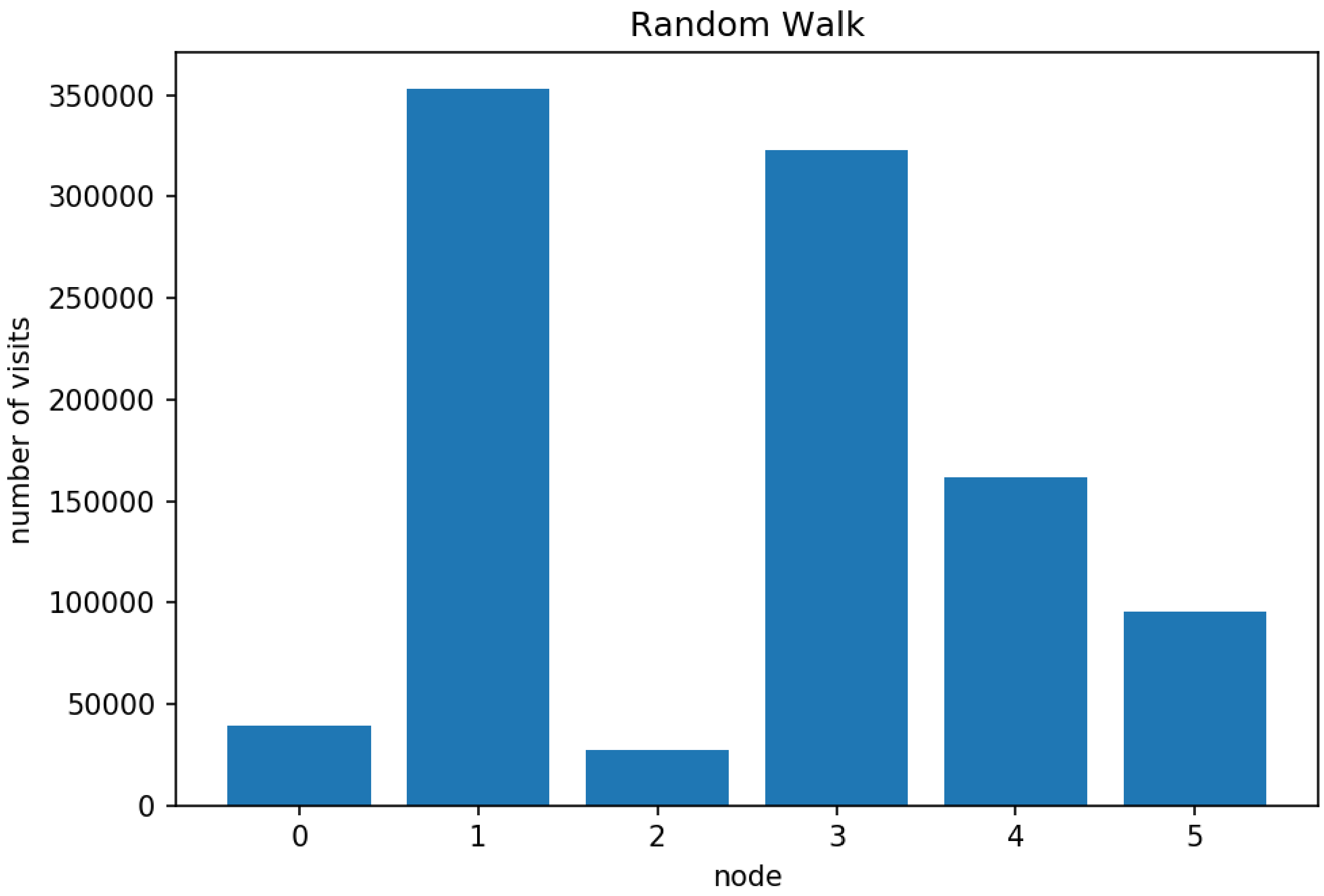
Python shell

```
| [[1], [3], [0, 1], [1, 4], [1, 5], [1]]
  [0.039371, 0.353392, 0.027766, 0.322108, 0.162076, 0.095287]
```

Simulate random walk (random surfer model)

pagerank.ipynb

```
import matplotlib.pyplot as plt
plt.bar(range(6), count)
plt.title("Random Walk")
plt.xlabel("node")
plt.ylabel("number of visits")
plt.show()
```



Transition matrix *A*

pagerank.ipynb

```
A = G / degree  # Normalize row sums to one. Note that 'degree'
                # is an n x 1 matrix, whereas G is an n x n matrix.
                # The elementwise division is repeated for each column of G
print(A)
```

Python shell

```
[[0.  1.  0.  0.  0.  0. ]
 [0.  0.  0.  1.  0.  0. ]
 [0.5 0.5 0.  0.  0.  0. ]
 [0.  0.5 0.  0.  0.5 0. ]
 [0.  0.5 0.  0.  0.  0.5]
 [0.  1.  0.  0.  0.  0. ]]
```

Repeated matrix multiplication

We now want to compute the probability $p^{(i)}_j$ to be in vertex j after i steps. Let $p^{(i)} = (p^{(i)}_0,\ldots,p^{(i)}_{n-1})$.

Initially we have $p^{(0)} = (1,0,\ldots,0)$.

We compute a matrix M, such that $p^{(i)} = M^i \cdot p^{(0)}$ (assuming $p^{(0)}$ is a column vector).

If we let $\mathbf{1}_n$ denote the $n \times n$ matrix with 1 in each entry, then M can be computed as:

$$p_j^{(i+1)} = \frac{1}{6}\frac{1}{n} + \frac{5}{6}\sum_k p_k^{(i)} A_{k,j}$$

$$p^{(i+1)} = M \cdot p^{(i)}$$

$$M = \frac{1}{6}\frac{1}{n}\mathbf{1}_n + \frac{5}{6}A^{\mathrm{T}}$$

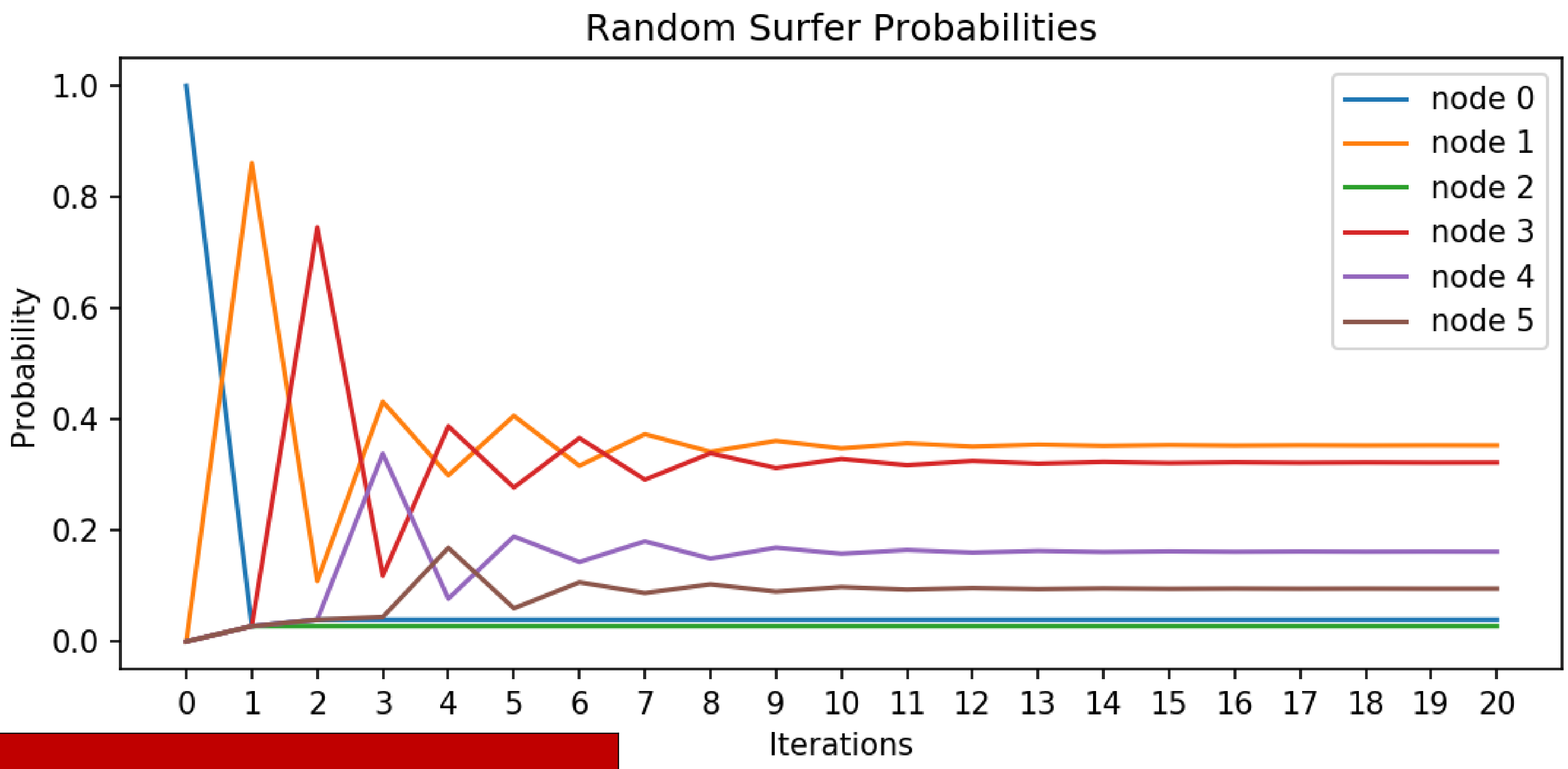
pagerank.ipynb

```python
ITERATIONS = 20
p_0 = np.zeros((n, 1))
p_0[0, 0] = 1.0
M = 1 / (6 * n) + 5 / 6 * A.T
p = p_0
prob = p   # 'prob' will contain each
           # computed 'p' as a new column
for _ in range(ITERATIONS):
    p = M @ p
    prob = np.append(prob, p, axis=1)
print(p)
```

Python shell

```
[[0.03935185]
 [0.35326184]
 [0.02777778]
 [0.32230071]
 [0.16198059]
 [0.09532722]]
```

Rate of convergence



pagerank.ipynb

```
x = range(ITERATIONS + 1)
for node in range(n):
    plt.plot(x, prob[node], label="node %s" % node)

plt.xticks(x)
plt.title("Random Surfer Probabilities")
plt.xlabel("Iterations")
plt.ylabel("Probability")
plt.legend()
plt.show()
```

Repeated squaring

$$\underbrace{M\cdot(\cdots(M\cdot(M}_{k\text{ multiplications, }k\text{ power of 2}}\cdot p^{(0)}))\cdots) = M^k\cdot p^{(0)} = M^{2^{\log k}}\cdot p^{(0)} = (\cdots(\overbrace{(M^2)^2)^2\cdots)^2}^{\log k}\cdot p^{(0)}$$

pagerank.ipynb

```
from math import ceil, log
MP = M
for _ in range(int(ceil(log(ITERATIONS + 1, 2)))):
    MP = MP @ MP
p = MP @ p_0
print(p)
```

Python shell

```
[[0.03935185]
 [0.35332637]
 [0.02777778]
 [0.32221711]
 [0.16203446]
 [0.09529243]]
```

PageRank : Computing eigenvector for λ = 1

- We want to find a vector p, with $|p| = 1$, where $Mp = p$, i.e. an *eigenvector* p for the eigenvalue $\lambda = 1$

pagerank.ipynb

```
eigenvalues, eigenvectors = np.linalg.eig(M)
idx = eigenvalues.argmax()  # find the largest eigenvalue (= 1)
p =  np.real(eigenvectors[:, idx])  # .real returns the real part of complex numbers
p /= p.sum()  # normalize p to have sum 1
print(p)
```

Python shell

```
| [0.03935185 0.3533267  0.02777778 0.32221669 0.16203473 0.09529225]
```

PageRank : Note on practicality

- In practice an explicit matrix for billions of nodes is infeassable, since the number of entries would be order of 10^{18}.
- Instead one has to work with sparse matrices (in Python modul `scipy.sparse`) and stay with repeated multiplication

Linear programming

scipy.optimize.linprog

- scipy.optimize.linprog can solve *linear programs* of the following form, where one wants to find a n x 1 vector x satisfying:

Minimize: $c^T \cdot x$

Subject to: $A_{ub} \cdot x \leq b_{ub}$
$A_{eq} \cdot x = b_{eq}$

dimension

$c : n \times 1$

$A_{ub} : m \times n$ $\quad b_{ub} : m \times 1$
$A_{eq} : k \times n$ $\quad b_{eq} : k \times 1$

NB: For industrial strength linear solvers, use solvers like Cplex or Gurobi

Linear programming example

Maximize

$3 \cdot x_1 + 2 \cdot x_2$

Subject to

$2 \cdot x_1 + 1 \cdot x_2 \leq 10$

$5 \cdot x_1 + 6 \cdot x_2 \geq 4$

$-3 \cdot x_1 + 7 \cdot x_2 = 8$

$\Updownarrow$

Minimize

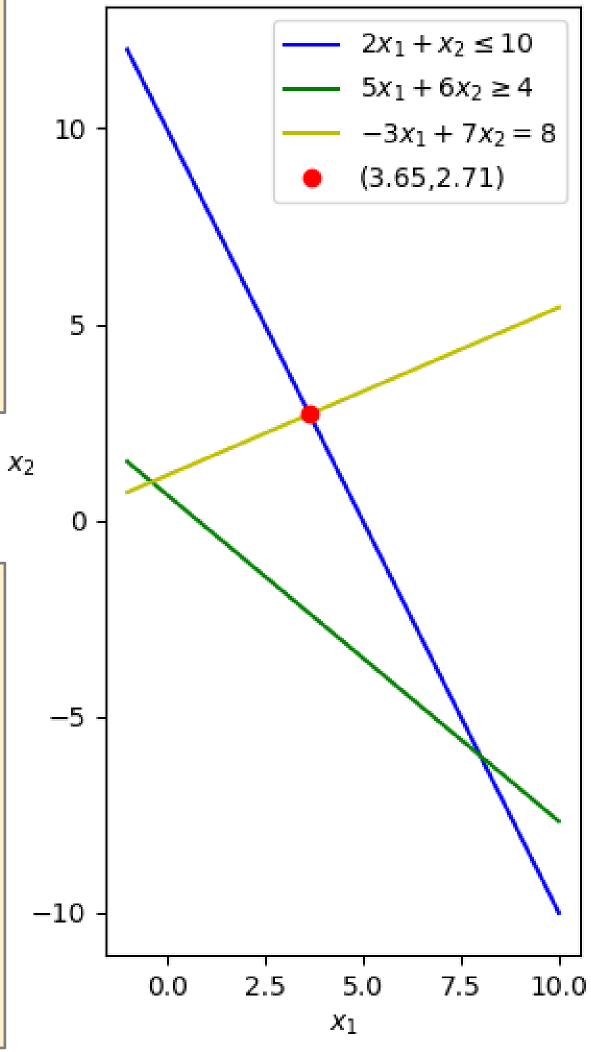
$-(3 \cdot x_1 + 2 \cdot x_2)$

Subject to

$2 \cdot x_1 + 1 \cdot x_2 \leq 10$

$-5 \cdot x_1 + -6 \cdot x_2 \leq -4$

$-3 \cdot x_1 + 7 \cdot x_2 = 8$



linear_programming.py

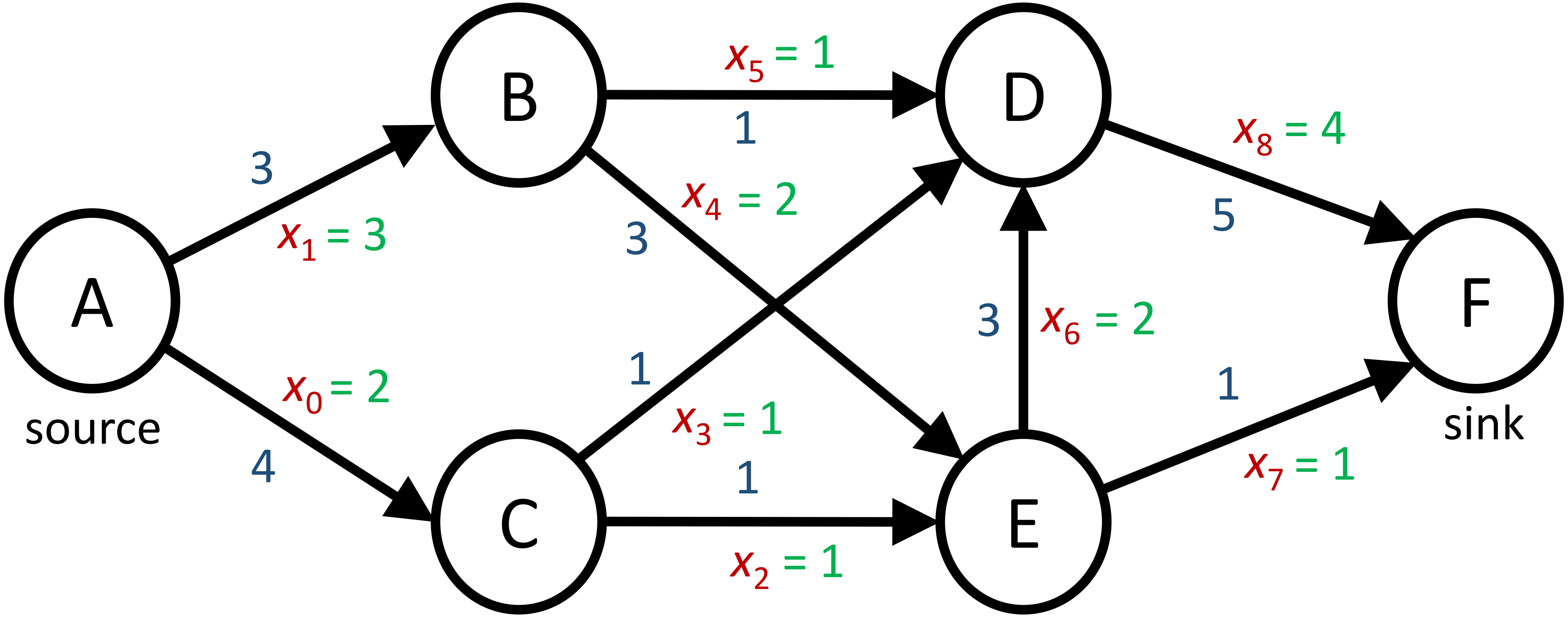
```
import numpy as np
from scipy.optimize import linprog
c = np.array([3, 2])
A_ub = np.array([[2, 1],
                 [-5, -6]])  # multiplied by -1 to get <=
b_ub = np.array([10, -4])
A_eq = np.array([[-3, 7]])
b_eq = np.array([8])
res = linprog(-c,  # maximize = minimize the negated
              A_ub=A_ub,
              b_ub=b_ub,
              A_eq=A_eq,
              b_eq=b_eq)
print(res)  # res.x is the optimal vector
```

Python shell

```
|    fun: -16.35294117647059
 message: 'Optimization terminated successfully.'
     nit: 3
   slack: array([ 0.        , 30.47058824])
  status: 0
 success: True
       x: array([3.64705882, 2.70588235])
```

Maxmium flow

Solving maximum flow using linear programming



Maximize

$x_7 + x_8$ flow value

Subject to

$$\left.\begin{array}{l} x_0 \le 4 \\ x_1 \le 3 \\ x_2 \le 1 \\ x_3 \le 1 \\ x_4 \le 3 \\ x_5 \le 1 \\ x_6 \le 3 \\ x_7 \le 1 \\ x_8 \le 5 \end{array}\right\} \text{capacity constraints}$$

$$\left.\begin{array}{l} x_1 = x_4 + x_5 \\ x_0 = x_2 + x_3 \\ x_3 + x_5 + x_6 = x_8 \\ x_2 + x_4 = x_6 + x_7 \end{array}\right\} \text{flow conservation}$$

Note: solution not unique

We will use the '[scipy.optimize.linprog]' function to solve the *maximum flow* problem on the above directed graph. We want to send as much *flow* from node A to node F. Edges are numbered 0..8 and each edge has a maximum *capacity*.

Solving maximum flow using linear programming

- x is a vector describing the flow along each edge
- c is a vector that to add the flow along the edges (7 and 8) to the sink (F), i.e. a function computing *the value of the flow*
- A_{ub} and b_{ub} is a set of *capacity constraints*, for each edge flow ≤ capacity
- A_{eq} and b_{eq} is a set of *flow conservation* constraints, for each non-source and non-sink node (B, C, D, E), requiring that the flow into equals the flow out of a node

Maximize

$\boxed{x_7 + x_8}$ $c^T \cdot x$ flow value

Subject to

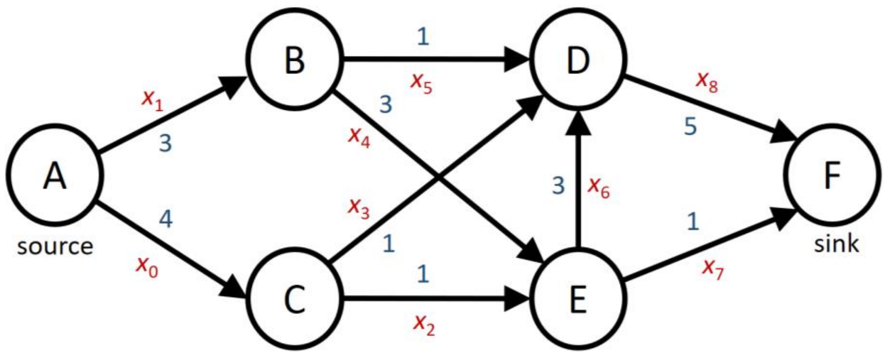
$$\left.\begin{array}{l} x_0 \leq 4 \\ x_1 \leq 3 \\ x_2 \leq 1 \\ x_3 \leq 1 \\ x_4 \leq 3 \\ x_5 \leq 1 \\ x_6 \leq 3 \\ x_7 \leq 1 \\ x_8 \leq 5 \end{array} \quad \begin{array}{c} A_{ub} \cdot x \leq b_{ub} \\ \Updownarrow \\ I \cdot x \leq \text{capacity} \end{array} \right\} \text{capacity constraints}$$

$A_{eq} \cdot x = b_{eq} = 0$

$$\left.\begin{array}{l} 0 = -x_1 + x_4 + x_5 \\ 0 = -x_0 + x_2 + x_3 \\ 0 = -x_3 - x_5 - x_6 + x_8 \\ 0 = -x_2 - x_4 + x_6 + x_7 \end{array}\right\} \text{flow conservation}$$

maximum-flow.py

```
import numpy as np
from scipy.optimize import linprog

edges = 9

#                           0   1   2   3   4   5   6   7   8
conservation = np.array([[ 0,-1, 0, 0, 1, 1, 0, 0, 0],   # B
                          [-1, 0, 1, 1, 0, 0, 0, 0, 0],   # C
                          [ 0, 0, 0,-1, 0,-1,-1, 0, 1],   # D
                          [ 0, 0,-1, 0,-1, 0, 1, 1, 0]]) # E

#                  0  1  2  3  4  5  6  7  8
sinks = np.array([0, 0, 0, 0, 0, 0, 0, 1, 1])

#                     0  1  2  3  4  5  6  7  8
capacity = np.array([4, 3, 1, 1, 3, 1, 3, 1, 5])

res = linprog(-sinks,
              A_eq=conservation,
              b_eq=np.zeros(conservation.shape[0]),
              A_ub=np.eye(edges),
              b_ub=capacity)

print(res)
```



Python shell

```
       fun: -5.0
   message: 'Optimization terminated successfully.'
       nit: 9
     slack: array([2., 0., 0., 0., 1., 0., 1., 0., 1.])
    status: 0
   success: True
         x: array([2., 3., 1., 1., 2., 1., 2., 1., 4.])
```