# Assignment: Graphs

Qianlang Chen (u1172983)

CS 5140 Spring 2021

## Problem 1

```
import numpy
from numpy import random
from scipy import linalg

M = numpy.loadtxt('./data/M.csv', delimiter=',')
n = M.shape[0]

def normalize(A): return A / linalg.norm(A, 1)

def matrix_power(M, t, q0):
    Mi = M
    for _ in range(t - 1): Mi = Mi @ M
    return Mi @ q0

def state_propagation(M, t, q0):
    qi = q0
    for _ in range(t): qi = M @ qi
    return qi

def random_walk(M, t0, t, q0):
```

```
    random.seed(0)
    i = q0.argmax() # index of the entry with a one
    for _ in range(t0):
        i = random.choice(range(len(q0)), p=normalize(M[:, i]))
    qs = numpy.zeros(len(q0)) # q-star
    for _ in range(t):
        i = random.choice(range(len(q0)), p=normalize(M[:, i]))
        qs[i] += 1
    return normalize(qs)

def eigen_analysis(M):
    eigvalues, eigvectors = linalg.eig(M)
    return abs(normalize(eigvectors[:, 0]))
```

## Part A

```
t = 2048
t0 = 100
q0 = numpy.array([1] + [0] * (n - 1)) # q0 = [1, 0, ...]
numpy.set_printoptions(formatter={'float': '{:.4f}'.format})
print('Matrix Power:')
print(matrix_power(M, t, q0))
print('\nState Propagation:')
print(state_propagation(M, t, q0))
print('\nRandom Walk:')
print(random_walk(M, t0, t, q0))
print('\nEigen Analysis:')
print(eigen_analysis(M))
```

```
Matrix Power:
[0.0483 0.0176 0.0414 0.0666 0.0586 0.1465 0.2350 0.0590 0.0562 0.2708]

State Propagation:
[0.0483 0.0176 0.0414 0.0666 0.0586 0.1465 0.2350 0.0590 0.0562 0.2708]

Random Walk:
[0.0430 0.0146 0.0449 0.0718 0.0576 0.1455 0.2344 0.0557 0.0527 0.2798]

Eigen Analysis:
[0.0483 0.0176 0.0414 0.0666 0.0586 0.1465 0.2350 0.0590 0.0562 0.2708]
```

## Part B

```
def find_closest(technique, M, t_range, q0, target_ans):
    min_err = float('inf')
    for t in t_range:
        ans = technique(M, t, q0)
        err = linalg.norm(ans - target_ans)
        if err < min_err:
            min_err = err
            best_t = t
            best_ans = ans
    return best_t, best_ans

q0 = numpy.array([1] + [0] * (n - 1)) # q0 = [1, 0, ...]
t = 2048
matrix_power_ans = matrix_power(M, t, q0)
state_propagation_ans = state_propagation(M, t, q0)

q0 = numpy.array([.1] * n) # q0 = [.1, .1, ...]
best_t, best_ans = find_closest(matrix_power, M, range(2049), q0,
                                matrix_power_ans)
print(f't = {best_t:,} worked best for Matrix Power:')
print(best_ans)

best_t, best_ans = find_closest(state_propagation, M, range(2049), q0,
                                state_propagation_ans)
print(f'\nt = {best_t:,} worked best for State Propagation:')
print(best_ans)
```

```
t = 69 worked best for Matrix Power:
[0.0483 0.0176 0.0414 0.0666 0.0586 0.1465 0.2350 0.0590 0.0562 0.2708]

t = 71 worked best for State Propagation:
[0.0483 0.0176 0.0414 0.0666 0.0586 0.1465 0.2350 0.0590 0.0562 0.2708]
```

## Part C

**Matrix Power**

- **Pro:** it only has to perform $O(\log t)$ matrix multiplications, which is useful when $M$ is small, or when we have multiple queries of start state ($q_0$) because we can pre-calculate $M^t$ only once.
- **Con:** matrix multiplication takes $O(n^3)$ for an $n \times n$ matrix, which makes it a bad option when $n > t$. In that case, one should use State Propagation instead, which has $O(n^2 \cdot t)$ running time.

**State Propagation**

- **Pro:** multiplying a matrix by a vector takes $O(n^2)$, which makes it a good option when $n$ is small.
- **Con:** if we had many queries of start state ($q_0$), we would have no other option but to run this algorithm many times. When the number of queries is big ($> n$), we should consider using Matrix Power to compute $M^t$ and cache it.

**Random Walk**

- **Pro:** taking a step (generating a random next step) only takes $O(n)$, which makes it very efficient when the matrix is big.
- **Con:** it only supports start states with a one in one entry and zeros in the rest. When we have start states not in that scheme, we should use Matrix Power or State Propagation instead, and the choice between the two depends on $n$ and $t$.

**Eigen-Analysis**

- **Pro:** it directly calculates the theoretical value for $q*$ (when $M$ is ergodic), which makes it the best choice when we want to know $q*$ precisely.
- **Con:** `linalg.eig` runs in $O(n^3)$. When the matrix is big, we should consider using State Propagation to get an approximation for $q*$ with less running time.

## Part D

```python
# Compute M**t for t = {6, 7}
Mi = M
for _ in range(5): Mi = Mi @ M
print('M**6:')
print(Mi)
print('All entries > 0:', all(all(x) for x in Mi))
Mi = Mi @ M
print('\nM**7:')
print(Mi)
print('All entries > 0:', all(all(x) for x in Mi))
```

```
M**6:
[[0.0372 0.0486 0.0462 0.0477 0.0472 0.0481 0.0437 0.0429 0.0513 0.0557]
 [0.0110 0.0131 0.0122 0.0206 0.0129 0.0213 0.0132 0.0169 0.0204 0.0215]
 [0.0347 0.0365 0.0338 0.0449 0.0353 0.0448 0.0370 0.0396 0.0442 0.0462]
 [0.0703 0.0625 0.0690 0.0571 0.0649 0.0631 0.0776 0.0532 0.0556 0.0662]
 [0.0651 0.0579 0.0507 0.0673 0.0494 0.0568 0.0566 0.0684 0.0647 0.0578]
 [0.1537 0.1543 0.1555 0.1567 0.1472 0.1367 0.1439 0.1649 0.1519 0.1431]
 [0.2514 0.2547 0.2434 0.2448 0.2567 0.2370 0.2301 0.2572 0.2381 0.2201]
 [0.0426 0.0496 0.0574 0.0545 0.0508 0.0588 0.0573 0.0467 0.0590 0.0700]
 [0.0562 0.0500 0.0611 0.0411 0.0547 0.0549 0.0665 0.0413 0.0481 0.0564]
 [0.2778 0.2729 0.2708 0.2654 0.2809 0.2785 0.2740 0.2689 0.2665 0.2629]]
All entries > 0: True

M**7:
[[0.0460 0.0474 0.0446 0.0514 0.0469 0.0506 0.0452 0.0495 0.0503 0.0497]
 [0.0195 0.0174 0.0152 0.0202 0.0148 0.0170 0.0170 0.0205 0.0194 0.0173]
 [0.0427 0.0404 0.0393 0.0441 0.0376 0.0405 0.0407 0.0445 0.0435 0.0416]
 [0.0592 0.0602 0.0687 0.0594 0.0620 0.0650 0.0696 0.0564 0.0640 0.0719]
 [0.0615 0.0617 0.0622 0.0627 0.0589 0.0547 0.0576 0.0660 0.0608 0.0572]
 [0.1480 0.1565 0.1494 0.1510 0.1567 0.1473 0.1413 0.1543 0.1499 0.1435]
 [0.2451 0.2433 0.2389 0.2347 0.2486 0.2396 0.2357 0.2415 0.2333 0.2251]
 [0.0540 0.0558 0.0536 0.0610 0.0531 0.0595 0.0566 0.0568 0.0613 0.0636]
 [0.0494 0.0498 0.0563 0.0501 0.0514 0.0547 0.0597 0.0446 0.0518 0.0615]
 [0.2747 0.2675 0.2718 0.2655 0.2701 0.2711 0.2766 0.2660 0.2658 0.2684]]
```

```
All entries > 0: True
```

According to *this document* (https://www.cs.princeton.edu/courses/archive/fall05/cos521/markov.pdf), a Markov Chain is ergodic if its transition matrix $M$ satisfies two conditions, connectivity and aperiodicity. Now, our matrix $M$ satisfies both conditions because

- It is connected: $M^t(i,j) > 0 \; \forall i,j$ when $t = 6$, and
- It is aperiodic: $\gcd\{t : M^t(i,j) > 0 \; \forall j\} = 1 \; \forall i$ because all entries in $M^6$ and $M^7$ are positive as well as 6 and 7 are coprimes.

Therefore, the Markov Chain represented by $M$ is ergodic.

## Part E

```
for i in range(n):
    if M[i, 5]:
        print(f'Node {i} can be reached from node 5 in one step with'
              f' probability {M[i, 5]:.1%}')
    else:
        print(f'Node {i} cannot be reached from node 5 in one step')
```

```
Node 0 cannot be reached from node 5 in one step
Node 1 cannot be reached from node 5 in one step
Node 2 can be reached from node 5 in one step with probability 10.0%
Node 3 can be reached from node 5 in one step with probability 10.0%
Node 4 can be reached from node 5 in one step with probability 40.0%
Node 5 cannot be reached from node 5 in one step
Node 6 cannot be reached from node 5 in one step
Node 7 cannot be reached from node 5 in one step
Node 8 cannot be reached from node 5 in one step
Node 9 can be reached from node 5 in one step with probability 40.0%
```