Asmt 6: Graphs

Anirudh Narasimhamurthy(u0941400)

April 29, 2015

1 Finding q_*

Part A

In this part of the problem we are asked to run the different methods of finding q_* and report the answers. The input matrix M given to us is 10 x 10 matrix. And $q_0 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$. We are also asked to run these different methods for t=512 iterations wherever applicable. The results of the different methods are described below:

Matrix Power method:

In Matrix power method, I coded M^{i+1} to be equal to $M^i * M$. And then applied $q_* = (M^t)q_0$. The result of q_* after running for t=512 is shown below:

$$\boxed{q_* = [0.0000, 0.0000, 0.1443, 0.2204, 0.2605, 0.1924, 0.0651, 0.0391, 0.0391, 0.0391]^T}$$

The result is 10 x 1 vector. This is the optimized answer that MATLAB provides when I print the result. When I looked into the values for the first two rows, the actual values were really small and the exact values are shown below:
$q_* = [$
7.78916082717592e-28,
1.02108887308038e-27,
0.144288577154309,
0.220440881763528,
0.260521042084169,
0.192384769539079,
0.0651302605210422,
0.0390781563126254,
0.0390781563126254,
0.0390781563126254
]

State Propagation method:

In state propagation method, we iterate $q_i + 1 = M * q_i$ for 512 iterations and the q value which we obtain after 512 iterations is our q_* value. The values are shown below:

$$\boxed{q_* = [0.0000, 0.0000, 0.1443, 0.2204, 0.2605, 0.1924, 0.0651, 0.0391, 0.0391, 0.0391]^T}$$

Again this is the optimized or rounded off annswer which MATLAB prints on the console. The exact values are shown below:

$q_* = [$
7.78916082717590e-28,
1.02108887308038e-27,
0.144288577154309,
0.220440881763527,
0.260521042084169,
0.192384769539078,
0.0651302605210422,
0.0390781563126253,
0.0390781563126253,
0.0390781563126253
]

Random Walk:

- In the random walk method we start with $q_0 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ and transition to a new state by choosing a new location proportional to the values in the i^th column of M. We iterate this for $t_0 = 50$ steps to obtain q'_0
- To implement this code, the logic which I followed was similar to that of K-means++ where we pick the centers proportional to probability of the square distance.
- I generated a random number u between 0 and 1 and for r running from 1 to 10, if the randomly generated value was less than the value of that particular column of M than I get that i value and put a 1 for q in that ith value and then start the next iteration. If my randomly generated value is greater than the column value I subtract the column value from the randomly generated value and then keep iterating.
- Since this is a randomized process, the results differ everytime when we run this code. The q'_0 value which was obtained after running for $t_0 = 50$ steps was :
  $q'_0 = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^T$
- After this the alogrithm was run for t=512 times and evry time I recorded the position where a 1 was recorded. The final q_* is a normalized version of the vector of these counts. In other words I had a counter which had the count of 1's at position 1, count of 1's at position 2,position 3 and so on till position 10. The normalized value is the result.
  The result q_* which was obtained for a particular random run was:
  $\boxed{q_* = [0, 0, 0.1504, 0.2285, 0.2441, 0.1934, 0.0625, 0.0430, 0.0391, 0.0391]^T}$
- This is fairly close to the result which we had obtained via Matrix Power and State Propagation method. However the important fact is that sum of this vector elements sums to 1 which is the important feature.
- In order for the result to be as close as possible, we could run the Random Walk say 10-15 times and then take the average of the results which we obtain every time since the process is randomized. The average should be fairly close to the answer which we get for Matrix Power, State Propagation and Eigen Anslysis

Eigen-Analysis:

In eigen analysis, I used the MATLAB function eig(M) to obtain the eigen vectors and eigen values. The eigen values are stored in a Diagnol matrix and the highest eigen value corresponds to the first entry in the matrix. We then need to take the corresponding first eigen vector and normalize it. The eigen vectors were stored in a matrix v in my code.

The result provided by MATLAB for the first eigen vector before normalizing it is shown below:
v(:1)= [0.0000, 0.0000, 0.3372, 0.5151, 0.6088, 0.4496, 0.1522, 0.0913, 0.0913, 0.0913]

I then normalized the eigen vector by dividing every element of the first eigen vector by the **1-norm value of the first eigen vector**. The reason for normalizing with 1-norm and not the default 2- norm is because of the Markov Chains property where we need to ensure $\boxed{\sum_i q[i] = 1}$.

The exact MATLAB code is shown below:

[v,d]=eig(M)
q_*=v(:,1) / norm(v(:,1),1)

And the resultant q_* that was obtained was :

$$\boxed{q_* = [0.0000, 0.0000, 0.1443, 0.2204, 0.2605, 0.1924, 0.0651, 0.0391, 0.0391, 0.0391]^T}$$

This is exactly the same result which we had obtained for the Matrix Power and State Propagation methods too and hence justifies that the answer is probably right.

Part B

Re-running Matrix Power and State Propagation method with different q_0

In this part of the question we are asked to run the above methods with different value of q_0 and report the t value for which our answer is as close to the true answer as the older initail state or in simpler terms the resulting $q*$ in Part A for the two methods.

New $q_0 = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]^T$

The closeness to the original result is defined by the threshold we define. I computed the difference by taking the **2-norm of the difference between old and new q* values** for each of the methods. That is
difference$_{MatrixPower} = norm(newq^* - oldq^*, 2)$ and
difference$_{StatePropagation} = norm(newq^* - oldq^*, 2)$

Threshold	**t value for Matrix Power**	**t value for State Propagation**
0.001	42	43
0.00001	82	83
0.0000001	124	125
0.000000001	172	173

Table 1: t-values for State Propagation and Matrix Power for different thresholds (2-Norm)

I have taken 3 to 4 threshold values and whenever my difference values was less than the Threshold, I wil report the t values for both methods for the specified threshold value below:

Another reasonable estimate for t would be based on the 1-norm distance too. The values of t when difference was calculated as the **1-norm** difference between two q_* values is shown in the table below:

Threshold	**t value for Matrix Power**	**t value for State Propagation**
0.001	49	50
0.00001	90	91
0.0000001	134	135
0.000000001	182	183

Table 2: t-values for State Propagation and Matrix Power for different thresholds (1-Norm)

Clearly we can see that the t-values for both methods are more or less close to each other barring one iteration and this is what I had expected intuitively to happen. It also feels right since the new $q0$ value makes these methods converge faster to the original result instead of having to do 512 iterations.

In ideal case, the thereshold difference which is considered a good enough limit is 10^7 which corresponds to 0.0000001. In that case the t-value which we get for 2-norm difference is 124 and 125 for Matrix Power and State Propagation method respectively.

Part C

Pros and Cons of each approach

Matrix Power:

Pros: Gives the correct solution for q_*. It would be the best option to use when the matrix M or P is relatively small. Since it is an iterative process and computes $M^t + 1$ from M^t we would be sure at every point if the chain is being cyclic and revolving around a fixed set of statesand determine if it would be Ergodic or not.

Cons: Each time M^tneeds to be computed to find the convergent value of state. And when the size of the Matrix is very big, then computing M^t will be computational overhead as it is a square matrix. We may even run into situations where cannot store all the values in the matrix or out of memory error. There will be an explosion in the size of the matrix as the number of states increases.

State Propagation:

Pros: States give the advantage of reaching convergence faster. Instead of having to compute M^T which is square matrix product operation, we can compute the product of smaller vector q and smaller matrix M. And using the brilliant property of Markov chains we can find $q_* = M * q_t$

Cons: Housekeeping all the states is an overhead. Also it is an iterative process and takes long time to complete.

Random Walk

Pros: Random walk finds convergence faster than based on the arbitrarily selected initial q_0 or q_0'. Since the selection q_0' is based on the probabilty proportional to the value, t=and the burn-in period helps pick a better q_0 for the problem. Hence in most cases this provides a good base and convergence might be faster. However there is no guarantee that it will always give the best solution but on an average it should be better than Matrix Power.

Cons: The process or the code implemented is inherently sequential. Hence one disdavantage is it is very hard to parallelize this method. Difficult to cover and find all the states based on the arbitrarily decided value of 't'.

Eigen-Analysis:

Pros: Faster solution based on eigen vector and optimized methods available in MATLAB for finding the eigen vector.

Cons: Does not always lead to an accurate solution and requires us to normalize the first eigen vector. Normalization procedure may vary and so the accuracy of results might differ. The second eigen value determines the rate of convergence and a higher value for it indicates slower convergence.

Part D

Is the Markov Chain ergodic ?

- **The given Markov Chain is not ergodic.** This is because for a Markov Chain to be ergodic there needs to be a 't' such that for all $n \geq t$ all the entries in P^n or in our case M^n should be positive. But clearly from the results of all the four methods, we found that the first two rows values in M^n for some large n were 0. So this violates the Ergodic property of Markov Chains.
- Another explanation is that from any starting position after t-steps there is always a chance that we are in every state, if the Markov Chain is ergodic. But clearly when we performed Random Walk experiment on the matrix, we found that the first two entries in q_* were zero indicating that those states were never reached even after 512 iterations.
- One more explanation is if we look at the Matrix M, assuming we are labelling all the rows and columns as A to J on both sides, we will find that from A we can only go to B or cycle back to A and from B we can go back to A only. So if we are at these states we kind of cycle back and forth and so it becomes cyclic. Thus the markov chain is not ergodic.

Hence for all the above mentioned reasons, **our Markov Chain is not ergodic.**

2 Bonus Question: TAXATION

- The trials in part 1.A can be repeated with taxation only for the State Propagation method. Because doing it for Matrix Power does not make sense as taxation dictates the probability of jump to a particular node which will correspond to q^t. And since we just obtain $q_* = M^t * q_0$ for Matrix power it is not useful.

- It is ideally implemented in State Propagation method. As an extension to question 1.D where we found the Markov chain not to be ergodic since not for all entries P^n or M^n were not positive for large enough n, our taxation will help us solve the problem.

  I implemented taxation to the State Propagation method by doing the following:

  1. Converted the original transitional probability matrix M or P to be the sum of the following matrices:
  $(1-\beta)M + \beta Q$, where Q is a n x n matrix with 1/n in each entry.

  2. This ensures that the entries in the first two rows of the matrix will be non-zero for higher values of m and potentially we could transition from one node to other node equally because of the higher probability values.

  Found the q_* value using $q_{i+1} = ((1-\beta)M + \beta Q)q_i$. Q in our case would be 10 x 10 matrix with all entries being 0.1. That is the transitional probability to every other state is equally likely. Multiplying it with $\beta = 0.85$ scales it up. Importantly multiplying the P matrix with $1-\beta$ scales all entries of M by 0.15 and the addition of the matrices ensures the original graph now becomes connected and the Markov Chain will now turn out to be Ergodic.

  The result obtained for q_* via Taxation after running for t=512 iterations is shown below:

  $$\boxed{q^* = [0.0544, 0.0682, 0.1119, 0.1489, 0.1857, 0.1288, 0.0897, 0.0693, 0.0693, 0.0739]^T}$$

  Comparing this with our original result for State Propagation we see that it was :

  $$\boxed{q_* = [0.0000, 0.0000, 0.1443, 0.2204, 0.2605, 0.1924, 0.0651, 0.0391, 0.0391, 0.0391]^T}$$

  Clearly we have come over the problem of connectivity and Markov chain not being ergodic via this Taxation scheme and the results are there to see as the first two rows are no longer zero and all the entries in P^n is positive.