Scientific Computation Project 2

Matthew Cowley, CID 1059624

March 2, 2020

Part 1

Throughout question 1 I have assumed a number of operations to be O(1): queue.popleft(), queue.append,len(queue), addition, subtraction, accessing elements in list and dictionary's.

1.1)

I have used an variant of breadth first search algorithm, with the following modifications it records the distance from the source and records how many distinct ways there are reaching each node on a rolling sum basis, using the node being explored as a root. Whenever a node is explored if the path equal to it then its distinct ways are added the score of the node. It then has a algorithm which works backwards to find the distinct path of O(N). I have implemented the BFS using deques objects for the que and dictionary's for efficiency.

BFS has complexity O(N+E) in the worst cases where all the E edges E and nodes N are searched. This altered variant uses similar complexity, there are 3 extra calculations(they realistic happen in constant time) when a node is being searched of a node and 2 extra if comparisons. One checks if a node is of minimum length from the source and so a distinct node. The other checks if the destination has been reached. This results in roughly 6N+E operations, E can at worst be N^2, so complexity is $O(N^2)$ for asymptotic running time (large N). But frequently the running time will be much better for this, especially for a sparse graphs.

1.2)

i) I have implemented an algorithm very similar to Dijkstra's, with the following modifications. It sets all nodes to unexplored apart from the start and with inf distance (takes N operations). When a node is searched all it's adjacent nodes are searched. A score for a node is based on whether it is greater than the node it is was searched from score. It also records how far the a node is from the source. It updates these scores until the destination has been reached or

no more unexplored nodes are connected to the source graph. These all take roughly 5 operations every time a node is explored, but are effectively constant time.

It always chooses the minimum score of the unexplored nodes to search next, here you would preferably store the distance using a binary heap. However using a dictionary for this result means it takes at worst $O(N)$ at every time a node is essentially explored. If a binary heap was used it would take $O(\log_2 N)$ constant time, with a Fibonacci heap you might get even better results.

After the graph has been explored there is a short algorithm to work out the path, working backwards, it is $O(N)$.

The asymptotic running time consists exploring all the nodes which is O(N) and then finding min every time step of unexplored nodes, giving an overall complexity of $O(N^2)$, with a binary heap you could improve this to $O(N \log_2 N)$

ii) It essentially the same algorithm but you are not comparing max of journey and all previous journeys but a standard Dijkstra's, with extra information stored at every time a node is explored and 2 more comparison's made(more elsif loops). It make sure if journey's scores are of equal distance, it chooses the one with fewer stops. (more elsif loops).

It has the same asymptotic running time of $O(N^2)$, as extra calculations will be constant in time. Again if you used a binary heap to store distance would more efficient to find the minimum giving $O(N \log_2 N)$.

1.3)

The algorithm I used for this problem uses BFS (you could have used DFS but both produce same result and we know nothing about the graph to be tested, so no advantage) to find a connected components and the stores the best cycle route for that then, moves onto the another component. It only stores the cheapest cycle route. It does this by storing a set of all the unexplored nodes and have a Que which is updated, from this set, when len(Que)=0. Within a connected component you have to store the 2 lowest leaving and arriving stations, in case the leaving and arriving station are the same, ensuring the cost is optimised. There are many comparisons but amount changes massively depending on the structure and values of graphs, but the worst it can be is 9 per node, how connected the graph is also effects results, altering temporary Que and set. However the worst case is a the same as BFS but with more comparisons, so has a asymptotic running time of O(E+N), which is $O(N^2)$.

Part 2

2.1)i)

For large N and small Nt,m, it uses a niave approach of 2 for loops and calling random choice to chose from adjacency list of current position for every time step. It does this for every walk.

The algorithm used for large M and Nt, is different, proving to be far quicker. It precalculates the random numbers setting them between 0 and 1, for all times and walk.s It also creates a condensed adjacency esc matrix of size Max degree$*N$, and a N-array of degree of nodes. It then uses the degree array to scale random numbers appropriately (with floor) and access a random node of dense condensed matrix, of appropriate degree. This requires only one for loop over time.

Each row i of the condensed matrix, is effectively an adjacency list of node i, plus some zeros which are never accessed. The data type of arrays possible is integer as well.

2.1)ii)

Using large M or studying the behavior over several times steps will give same results as all simulations have the same initial conditions. As we can see the

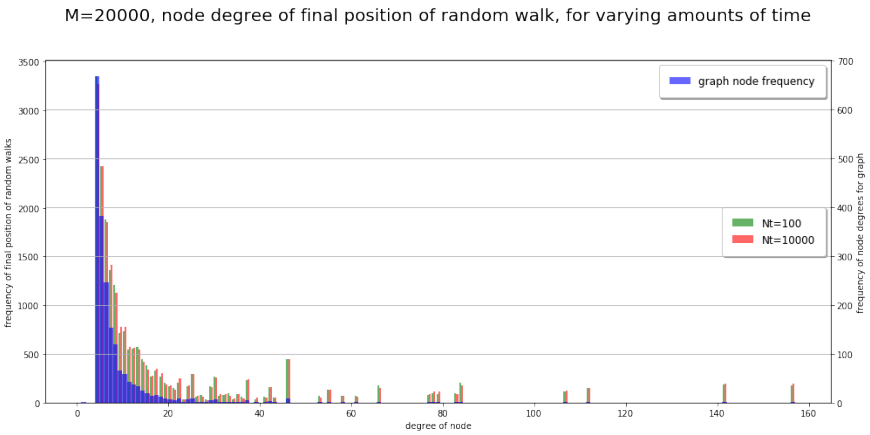

Figure 1: Frequency of final position of simulations wrt to degree, compared to frequency of the nodes of that degree

graph above shows, there is not significant difference between Nt=10000 and Nt=100, suggesting a sense of stability/pattern. It also appears that the higher the degree the proportion of walks relative to frequency of nodes of that degree. The figure below plots the frequency of the final position, divided the frequency of the nodes of that degree and the number of simulations, to give an average.

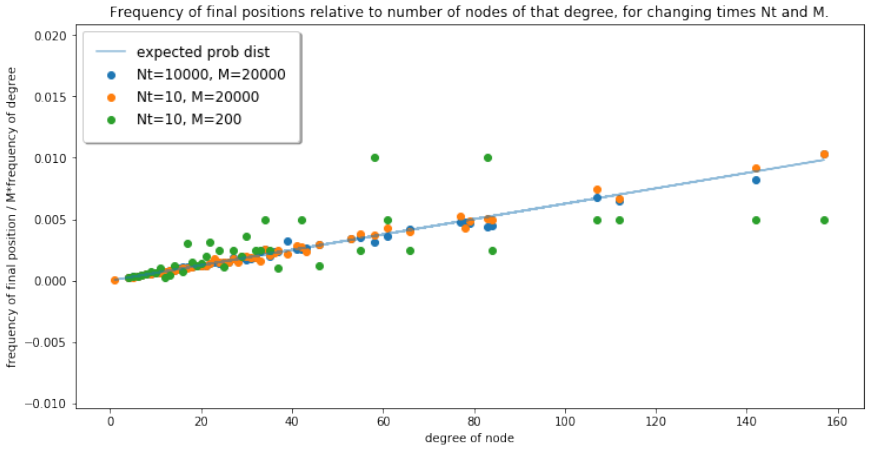

Figure 2: How nodes of higher degree, are visited more frequently

As we can clearly see nodes of higher degree are visited more frequently, which makes sense give the nature of the random walk. Interesting to see that for low Nt and M the results are more varied, if either is large and we report a stable average. We can view these random walks as Markov chains as they do not depend on previous time steps. Here the probability distribution is the following:

$$\frac{\text{Node degree}}{2 * \text{Sum of the edges}}$$

We expect the simulations to tend to the stationary distribution (the limiting distribution in this case) of the Markov Chain for large t .

2.1)iii)

Keypoints:

- First eigenvalues is approximately zero for all operaters and all eigenvalues are negative (which are scaled by D).
- We expect L to spread out evenly across the nodes, L_s to favour nodes of lower degree and L_s^T favours nodes of of high degree similar to Random walks model.
- Simulations to support these conclusions.

As we can see all eigenvalues are negative apart from the first, not shown is how the Laplace operaters becomes very negative. This is expected as the Laplacian is a postive semi definite matrix (so -D gives negative eigenvalues) . When solving the linear diffusion it becomes an eigenvalue problem (essentially solutions are of the form $w \exp(-\lambda t)$ where w is eigenvector and λ is an

eigenvalue.). So for asymptotic behavior we expect all operators to produces solutions which are stable, apart from maybe characterises of the first eigenvalue. I will not take an interest in D as it just scales the eigenvalues, so speeds up the rate of convergence, but does not change behavior of the model, apart from when D is very small.

The eigenvector corresponding to a eigenvalue 0 is a form of the initial conditions.

The first eigenvalue for each operator is are of the following sizes $2.38239199 \times 10^{-14}, -4.68891113 \times 10^{-17}, -4.39065381 \times 10^{-16}$, for the Laplacian L, scaled Laplacian L_s and L_s^T respectively. So all are approximately zero. In the case of L, all the eigenvector is just ones and so we expect all paths to be split equally among the nodes, which is just the end result of linear diffusion. Which is not what we expect from the random walks model.

When we look at L_s the main item producing dynamics the $DQ^{-1}A$, which is the adjacency matrix with row i multiplied by 1/degree i, so there will be a greater change in i for nodes with lower degrees. Perhaps giving a random walk model such as $p_{ij} = A_{ij}/q_j$. The transpose of L_s gives us the the dynamics we are seeking, higher node degree reducing rate of change to other nodes, which leeds to the the walks favouring nodes of high degree.

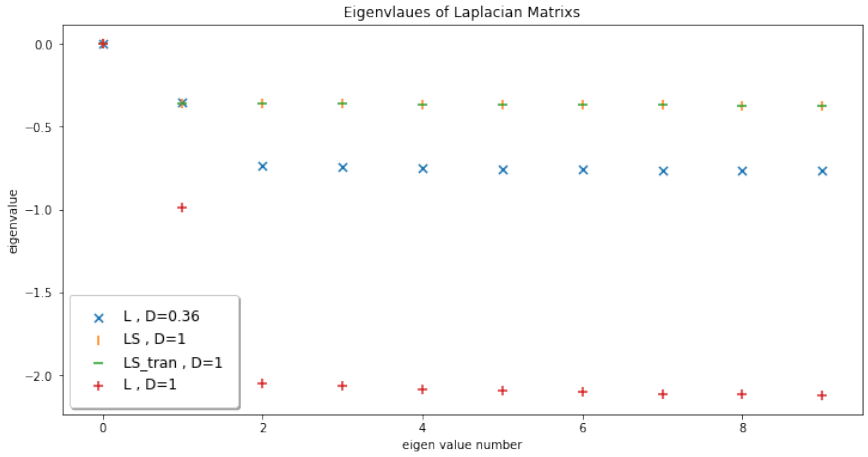

Figure 3: Eigenvalues of the different operaters (multiplied by -D)

When we look at L_s, the main operator producing dynamics is $DQ^{-1}A$, which is the adjacency matrix with row i multiplied by 1/degree i, so there will be a greater change in i for nodes with lower degrees. Perhaps giving a random walk model such as $p_{ij} = A_{ij}/q_j$. The transpose of L_s gives us the the dynamics we are seeking, higher node degree reducing rate of change to other nodes, which leeds to the the walks favouring nodes of high degree.

Here we can clearly see L_s^T exhibiting very clearly similar dynamics to the random walks, with high Nt and M simulations converging (still some randomness exhibited). The L is spread out evenly among the nodes,shown by it low frequency value. L_s is a bit unexpected it is has a surprisingly high frequency, and does not favour the lower degree nodes, this maybe be because the highly connected and the effect of the low degree's balancing out.

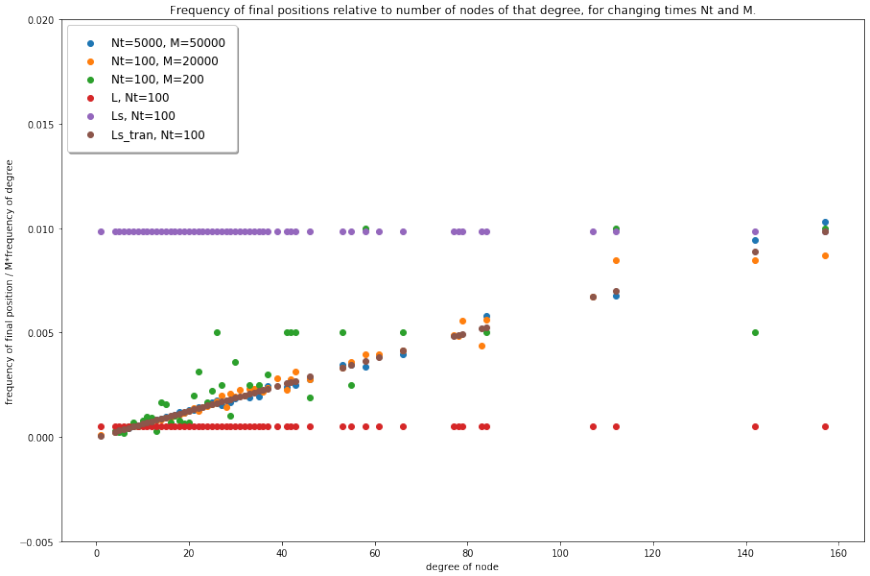

Figure 4: How the operators compare to the random walk simulations

The frequency balancing out for Ls is also shown below, but here we can see initially the frequency is high and after a long period of time it different nodes converge to the same frequency even though they have different degrees.

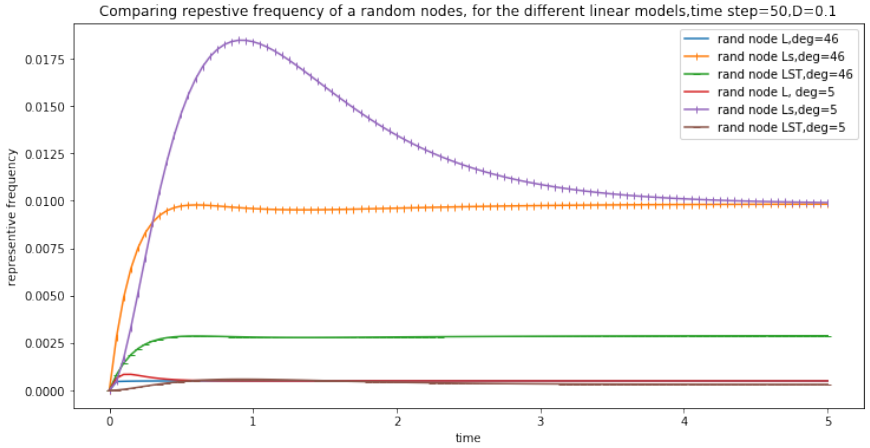

Figure 5: Looking at specific nodes

2.2)i)

The main bulk of the work for RHS is $A_{jk}i_k$ calculation, which I have implemented using sparse matrix multiplication in scipy, the number of operations doing this means it is 2E operations, where 2E is number of edges in graph, instead of N^2. I have precomputed A to include γ as well. You have ones vector minus y, and then element-wise vector multiplication with vector $A_{jk}i_k$ which is 2N. you then add this to $-i_j$ element wise which is another 2N. So the number of operations for one run of RHS is 2E+4N.

2.2)ii)

I have used all the parameters suggested in the questions, with D=0.1 , with initial condition of 1 at node of max degree of the graph. The models are all tested on a Barabasi-Albert graph of size 100 and with m=5, a property of these graphs is that they are always connected, as it built by adding nodes to graph and is connecting them nodes already in the graph (in this case 5). I have left all the figures for this section at the end as there are many of them. Keypoints:

- Model A displays wavelike behavior.
- Model B creates a stable but oscillating solution.
- Linear model is just the diffusion of initial conditions through the network.

Linear Model (fig 6)

Here clearly all the intensity's converge to one intensity, spreading out initially in a diffusive a nature, so from nodes of high intensity to connected low intensity's.

Model A (fig 7)

Model A is the infectious disease model covered in lectures. β is kind of linear decay of infection or traffic, wheres as links with other nodes are represented by $A_{jk}i_k$ with more connected nodes resulting in a linear greater change in a node, but this is limited by $1 - i$ term, so the higher the traffic of a node the less effect other nodes have. But overall we should wave like behavior, so if given a perturbation it should spread out like a wave and then reach a form of equilibrium where rate of recovery is the same as the rate of spreading. We can clearly see this happening in the simulation.

The wave like behavior is also represented in variance and mean graphs.

Model B (fig 8)

Model B effectively represents a 2nd order diffusion process of a particle i, which can be represented as follows:

$$\frac{d^2 i_j}{dt^2} = \sum_{k=0}^{N-1} \alpha L_{jk}(i_k - i_j)$$

the i_j term also disappears as when you sum over k, the rows of the Laplacian goes sum to zero so the i_j disappears, so:

$$\sum_{k=0}^{N-1} L_j k i_j = 0$$

therefore model B is:

$$\frac{d^2 i_j}{dt^2} = \sum_{k=0}^{N-1} \alpha L_{jk} i_k$$

Which indicates that the rate of change of rate of change, is equal to the number of connections of a given node and the values at those nodes, it is similar to the acceleration of particles changing for each node in the graph.

For this problem we are could take the view point that we are trying to solve $y'' + cy = 0$ for every node for some $c > 0$ (α is always negative) and boundary conditions depending on initial conditions and other values of the graph. This has a solution:

$$y(x) = c_1 \cos\left(\sqrt{\lambda}\, x\right) + c_2 \sin\left(\sqrt{\lambda}\, x\right)$$

with c_1 and c_2 depending on these boundary conditions and other values of nodes essentially. This could explain why we are getting lots of oscillatory

behavior in figure 8. The oscillatory behavior can also be linked to the eigen values with the first 2 being equal to zero and the rest being complex, the complex indicate oscillatory behaviour and the 2nd eigenvalue being zero could be linked to this 'changing acceleration' or rate of rate of change, changing. This is again rein-enforced when we look at variance of the model in figure 10, as it is oscillating.

Comparisons

All the models are very different, for example when you look at the variance of figure 10 they are all very different. Model A variance stables once the distribution stabilises, linear model just decreases as expected and Model B is constantly oscillating. However figure 9 suggests that maybe Linear L and Model B are the same, however after further investigation Linear model tends to initial condition divided by number of edges, where as Model Bs varys between 0.01 and 0.009, which appears to maybe be the initial condition divided by the number of points.

Figures for part 2.2

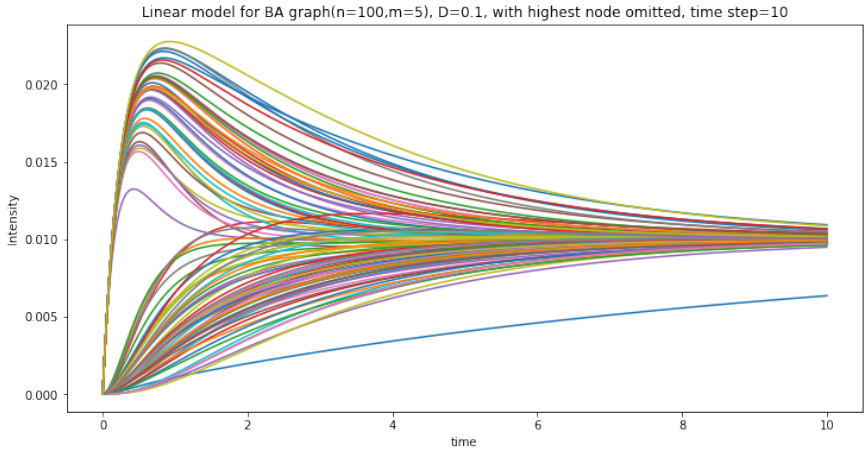

Figure 6: I have ommited the node of highest degree, as it disrupts the scale of the graph, it just decreases in a exponential like manner to the same convergence of other nodes

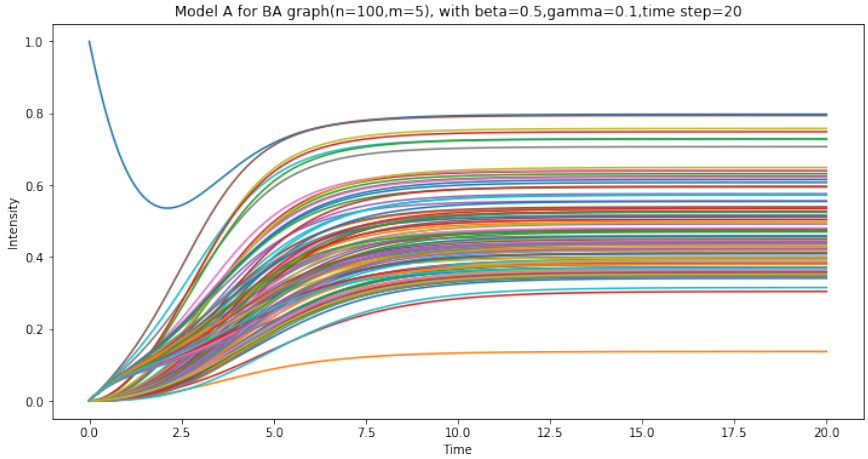

Figure 7: Model A simulation

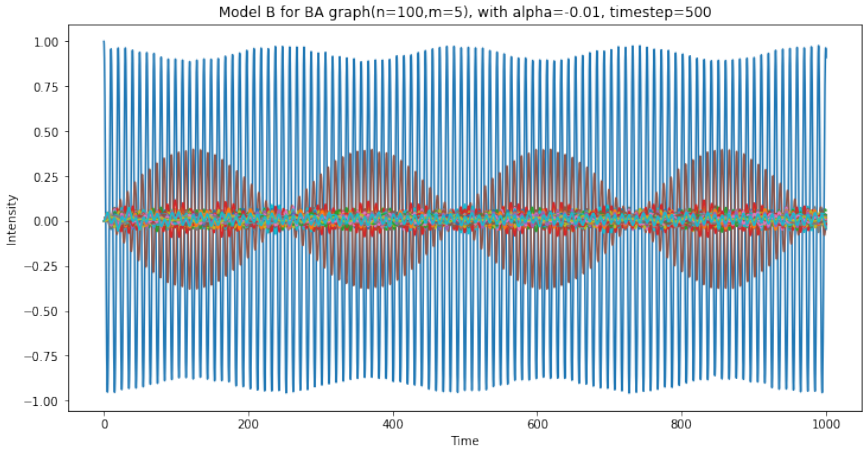

Figure 8: Model B simulation

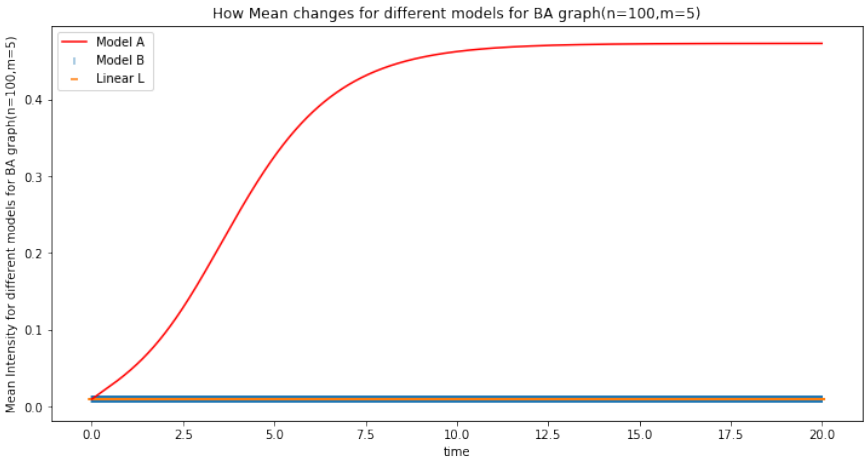

Figure 9: Means of the different models

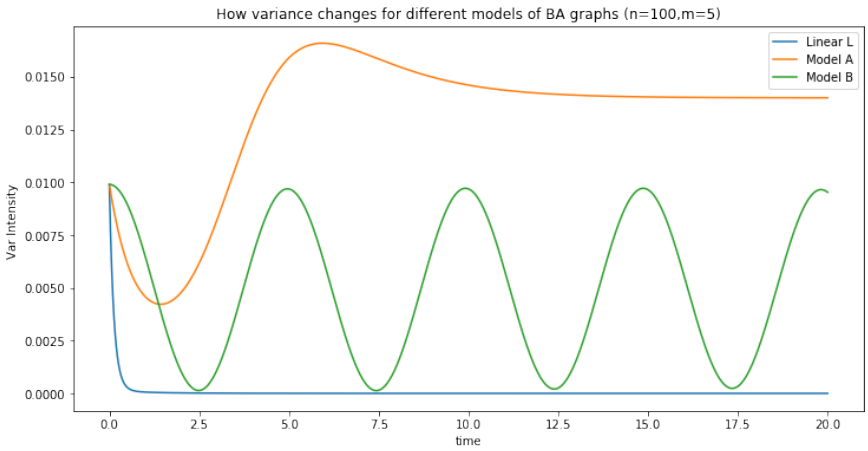

Figure 10: Variance of the models