Scientific Computation Project 2

Jonathan Cheung 01063446

March 2, 2020

Part 1

1.1)

The function 'flightLegs' takes as its input, an adjacency list, starting, and destination airport from an unweighted network of airports defined from the adjacency list. It uses a breadth first search for the shortest paths. By its nature, this breadth first search will find the shortest journeys first, since it branches through the network one layer at a time. Thus, the first path it finds will be the shortest. So we have designed our algorithm to always record the first path it finds, and continue to record paths until a path has a length greater than some previous path element. The time complexity of this algorithm, using an adjacency list, is O(N+E), where N is the number of nodes and E is the number of edges. We can at worst case only visit each node once since we track explored nodes. We also at worst case visit each edge twice since each edge has two nodes. Depending on the graph N or E can be much larger than the other and in this case we have O(M) where M is the max(N,E). For efficiency, we have used a deque, which can use popleft() to remove elements from the queue at O(1) time, compared to using pop(0) on a list which is O(n).

1.2i)

The function 'safeJourney' takes the same inputs as flightLegs; it takes an adjacency list, start, and destination node. However the input adjacency list is slightly different since it contains a safety factor. The lower the S value, the riskier the route. The function will find the safest journey, by finding the route with the smallest maximum S (risk) value. In other words, it finds a route such that the most dangerous part of the route is the least dangerous. The algorithm works by using a modified Dijkstra's algorithm. However, instead of summing the weights, it takes the maximum and minimum weights. Dijkstra's algorithm in this case has a time complexity of $O(N^2)$ where N is the number of nodes. We could speed with up to O(Elog(n)) if we implemented a binary heap.

1.2ii)

The function 'shortJourney' takes the same inputs as 'safeJourney'. Though in this Alist, the weights represent time of the journey. The function will output the shortest journey. Again, this function uses Dijkstra's algorithim. The code is very similar to 1.2i). The key difference is that we are summing the weights, not taking the maximum. This has the same time complexity as above since it is still Dijkstra's algorithm.

1.3)

The function 'cheapCycling' takes two inputs: SList and CList. SList is a list of two element lists. The ith element of the list corresponds to the arrival and departure cost for the ith station. Clist is an adjacency list to show which stations you can cycle to and from. The function outputs a list of two element lists. The ith element tells you which train stations to make your arrival and departure journeys to and from. This function works by creating a dictionary of all the nodes. It then extracts a single node and explores all the nodes and seeks the minimum arrival/departure costs in a BFS manner. The difference from a normal BFS is that it removes the explored nodes from both the queue AND the dictionary of all points. This means that if the BFS completes searching a connected component, the queue will be empty, but the dictionary will contain all the remaining nodes that are in the network, but not in this just completed connected component. As a result of this, the algorithm can then extract an element of this dictionary and then run a new BFS search on a new connected component. We have used a deque for the queue since it is O(1) to popleft() compared to O(n) to pop(0) of a list. We have used a dictionary for the overarching set of unexplored nodes. This is because, it is O(1) to search for a node in a dictionary, and we have to search for the node that we pop from the deque since the node wont necessarily be located as the first element. Another thing to note is that we have to reset the minimum cost to arbitrarily large number every time we search a new connected component. The time complexity of this algorithm is still O(N+E), where N is number of nodes and E is number of edges. despite having two while loops, this has the same time complexity as a BFS search. This is because the two while loops are linked; nodes are removed from both of the sets that the while loop checks for emptiness.

Part 2

2.1)

To simulaute M Nt-step random walks on a Networkx graph, we can use function 'rwgraph'. This function has inputs:G (a network x graph), i0 (starting node), M(number of walkers), Nt(time steps). The function outputs these results as a M by Nt array where each column represents a single time step and each row represents a single walker. This function fills in the first column using the initial conditions. The first time step is calculated by using numpy.random.choice on the neighbours of this initial node to generate M random nodes to fill in this first time step. To calculate the remaining walker positions, we first fill in the rest of the matrix with a random number between 0 and 1. In this same file there is a function 'nextboi' which takes as its input a node position of a walker, and a decimal value between 0 and 1. It calculates the number of neighbours that the node has and uses this decimal to decide which node to pick. For example, if a node has 4 neighbours, then there are 4 equally sized partitions between 0 and 1, with 0.25, 0.5, and 0.75 dividing these partitions. If a decimal value is 0.6, it will fit into the 3rd partition so the third node is chosen.

The figures 1-4 have been plotted by looking at the final positions of the walkers and plotting the number of walkers on each node. We have taken M to be significantly larger than the n=2000 (the number of nodes) to ensure that every node is more likely to have walkers on it. We also plot the degree of each node against the node itself. These graphs are plotted as linegraphs so we can compare the shapes of these practical results against what we expect theoretically. A slight problem is that the practical results has 8000 walkers, but the sum of the degrees for each node in our graph is not the same. So we can not compare the number of walkers on a node to the degree. However, we can compare the shapes of these line graphs since the y-axis is a linear scale in both. We can see that the line is converging to the same shape as the degree node plot as t increases (figs). The difference in shapes at large t=200 can be attributed to the randomness of the rwgraph simulation.

For the Laplacian, scaled Laplacian, and transposed scaled Laplacian, we can find the limiting distribution. This is i(t) as t tends to infinity. Unfortunately, I have been unable to get my code to calculate the matrix exponential (see rwgraph_analyze2). However, I believe that the normal laplacian and scaled laplacians should have similar limiting distributions.

2.2)

We have functions 'modelA' and 'modelB' that simulate the models defined in the assignment. The function 'transport' will return plots for these functions. Fig 5 shows the modelA simulation for the node with highest degree and one other node. We can see that the highest degree node has been set with 0.1

initial value. We can see this concentration decrease very slightly at the start since the concentration is being spread out to the neighbours, however this node has highest degree, so the concentration eventually returns and this node will increase and remain higher than all other nodes (fig 6). These concentrations eventually reach an equilibrium point and you can see that this happens at around time=150. The initial node we chose has the highest concentration value at equilibrium point. Perhaps this is because it has the highest degree and so it needs a high concentration value to match the concentrations of all neighbouring nodes.

If we look at figures 7 and 8. We can see how this concentration changes when gamma or beta are increased. Increasing gamma causes the increase in concentration to be more steep. In the context of a disease perhaps this represents how infectious a disease is. Increasing beta causes the increase to slow down and each node to take on more differing values. For a disease perhaps this represents how strong the populations immune system is and can resist being infected.

For modelB, we look to figures 9-10. These figures represent the s and i solutions. The oscillatory behaviour suggests that the system moves periodically through the network. In the graph for i (fig 9), we can see that the oscillation is centred at the zero axis. So perhaps this system could be a crude representation of wealth and how when one individual becomes wealthier, it is at the expense of another. The initial node is an individual with the most connections and so has the most income and expenditure so this node can be seen to have the largest oscillations. It also appears that the some of the other nodes have increasing magnitude of oscillations. Perhaps, this shows overall increase of wealth in the system, but it is not clear to me why this is. Given more time I would like to investigate this. Looking at fig 10, this is a plot of the s equation. I believe this could represent flux and it is increasing over time for some reason related to the how some nodes in I have increasing magnitude.

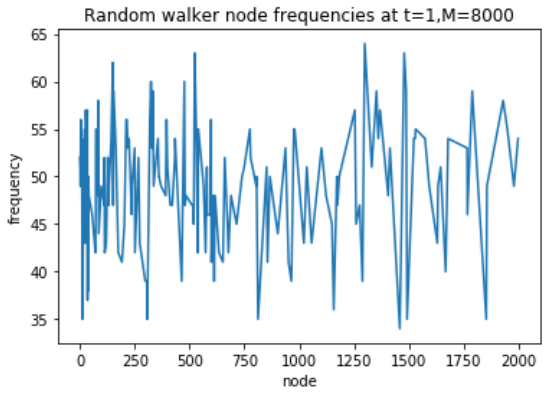

Figure 1: Figure for question 2.1

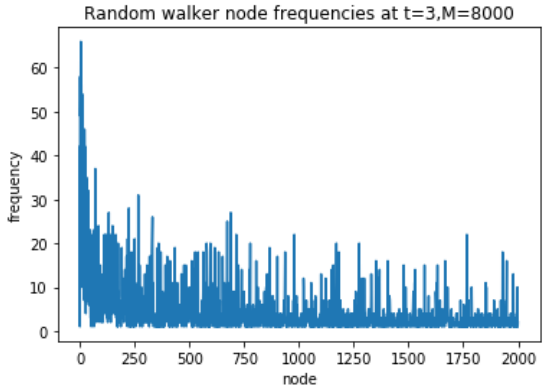

Figure 2: Figure for question 2.1

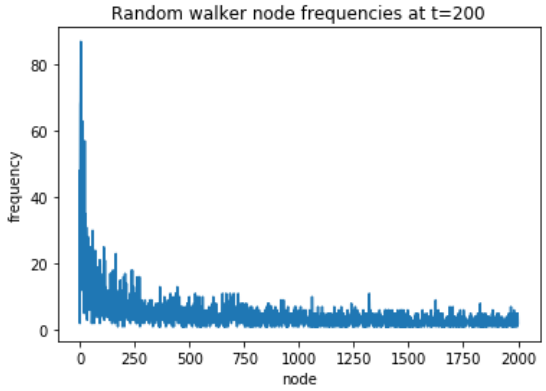

Figure 3: Figure for question 2.1

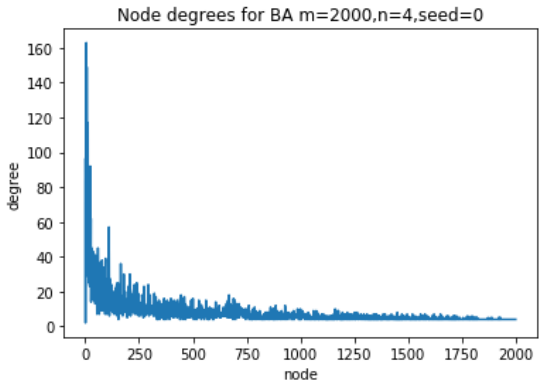

Figure 4: Figure for question 2.1

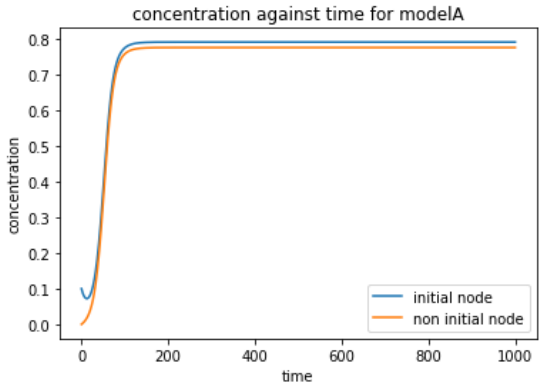

Figure 5: Figure for question 2.2

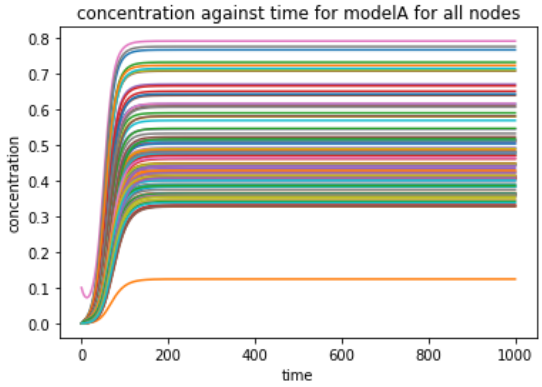

Figure 6: Figure for question 2.2

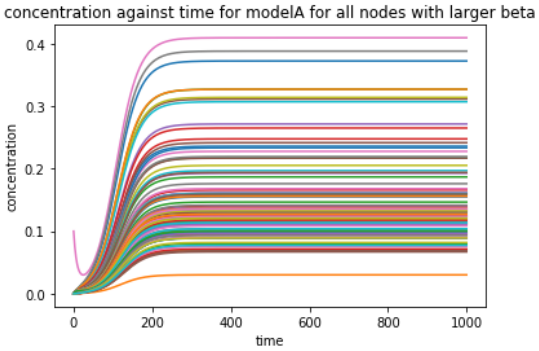

Figure 7: Figure for question 2.2

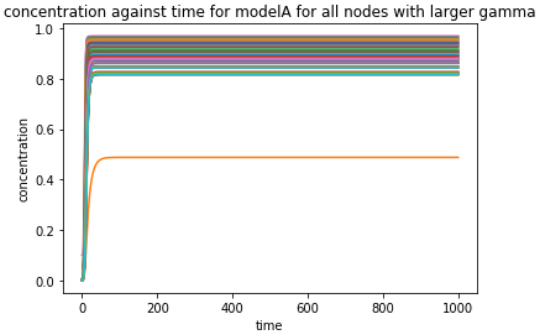

Figure 8: Figure for question 2.2

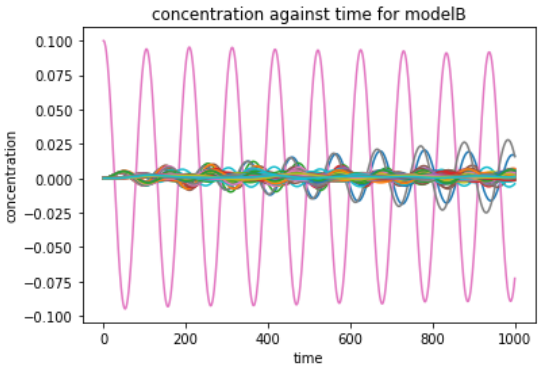

Figure 9: Figure for question 2.2

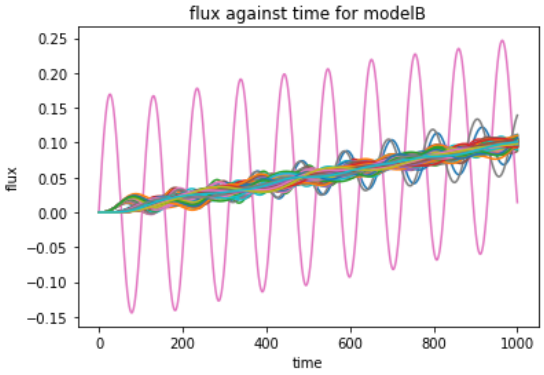

Figure 10: Figure for question 2.2