Scientific Computation Project 2

Marwan Riach, CID: 01349928

March 2, 2020

Part 1

For this part of the task, I will refer to $|E|$ to be the number of edges of my graph and $|V|$ to be the number of nodes (or the length of Alist or Clist).

1.1)

In 1.1, I use a modified version of breadth-first search. BFS is suitable in this instance because I want to find the minimum distance between my start node and my destination node, with the distance being equal to 1 and the metric being the number of flights. In the nth iteration of my BFS, I discover all nodes that are n flights away from the source node, labelling the nodes 1 and recording their distance from the source node as I go along. If during my nth iteration, I hit my destination node, then I know that the shortest distance between my source node and my destination node is n and hence I terminate the search from there.

In order to compute the number of distinct paths to my destination from my source, I keep a tally as I go along of the number of distinct paths to each node for all levels. If I discover an unvisited node Y from my node X, then I add the number of distinct paths to X to the number of distinct nodes to Y. If from my node X, I encounter a node, Z whose distance is 1 greater than that of X, I also add the number of paths to X to the number of paths Z. This ensures that I have a rolling sum of the number of possible paths to each node for all layers in my graph.

To improve the efficiency of this algorithm and reduce the running time, I used the *deque* object from the *collections* library so that when an item is removed from the front of the queue, the process is of order 1 rather than order n. Furthermore, I also terminate the BFS when I've reached the depth of my destination node, so that I do not have to search any further. This doesn't reduce the worst case running time, however it does reduce the running time on average.

The asymptotic running time is $\mathcal{O}(|E| + |V|)$. In the worst case, I iterate over all my nodes until the final one is the destination and the graph is not at

all sparse. Note also that $\mathcal{O}(|E|)$ lies in between $\mathcal{O}(1)$ and $\mathcal{O}(|V|^2)$. This means my worst case asymptotic running time is $\mathcal{O}(|V|^2)$.

1.2.i)

For this algorithm, I use a modification of Dijkstra's algorithm. Whereas in standard Dijkstra's algorithm, you compute the shortest distance to all nodes from your source, here I am computing the minimum d such that all edges traversed have no values greater than d. To implement this, I mark a node as visited once it is the node with the smallest d value associated to it. If this visited node is my target node, then I can break the algorithm which returns the path and safety factor, or empty vector if no such path exists. The path is stored as I visit each node efficiently. I store the node visited prior to the node I mark visited so that I don't have to store lots of paths which is both time and space inefficient; this allows me to retrace the steps of my destination node and only return the path I want.

Ideally in this algorithm, I would have used *heapq* since this would have ensured my storing each value and finding the minimum value of d and its corresponding node would have been $\mathcal{O}(\log|V|)$ (base 2 logarithm) as opposed to $\mathcal{O}(|V|)$; therefore, in my code, I would have used a binary heap instead of a dictionary to store the unexplored nodes, from which I would extract the minimum.

Efficiency wise, my algorithm makes use of dictionaries instead of lists in order to locate nodes since they are called by their key and if a list were used I would have to constantly re-index everything for each time I removed an element (Order n rather than order 1). My algorithm also checks if the minimum node is equal to the initial large value set. This is to avoid having to check more than one disconnected node before I deduce the destination is not reachable from the start.

Complexity wise, the asymptotic complexity is $\mathcal{O}(|E| + |V|^2)$ since this is equivalent to the complexity of Dijsktra's without a binary heap (from lectures). Had *heapq* been used, this complexity would have reduced to $\mathcal{O}((|E| + |V|)\log|V|)$.

1.2.ii)

This is identical to the algorithm for 1.2.i except the rolling sum of time is compared and the minimum time is stored instead and comparisons are made to ensure that the journey with the shortest time also goes through the fewest nodes. This is measured by comparing the lengths of the paths to the same node with the same time and updating accordingly. To make sure no extra cost is added to the running time (e.g. copying the path) I introduce a third element in my dictionaries which is the shortest distance from the source node.

Again, a binary heap could be used here in an identical way to 1.2.i to improve the efficiency by using priority queuing which leads to a logarithmic time complexity as opposed to a linear one.

The complexity of this algorithm is identical to the complexity of 1.2.i since comparing the length of an object is $\mathcal{O}(1)$. I.e. the complexity is $\mathcal{O}(|E| + |V|^2)$.

1.3)

In this algorithm, I make use of BFS in order to find the sets of connected nodes. For each set of connected nodes I also store the going in price and going out cost. For isolated nodes, I skip them by checking if their entry in Clist is empty.

Once I have my sets of connected nodes, I store the 2 nodes which have the smallest going in cost and the 2 nodes which have the smallest going out cost. I then check if the smallest in and smallest out are the same node. If this is the case, then I compare the 2nd smallest going in with the smallest going out and vice versa: exhaustively this will give me the smallest possible sum in each set of connected nodes. When this is computed, I compare the sum with the rolling sum and if this sum is less than the rolling sum, I update the rolling sum and the nodes corresponding it. This is repeated until all sets of connected nodes have been iterated through.

I store the nodes corresponding to the smallest values as I go along to prevent superfluous calculations. In addition to this, I don't make any further comparisons if the minimum going in and minimum going out correspond to different nodes.

In terms of complexity, the first part of the algorithm which corresponds to BFS is $\mathcal{O}(|E| + |V|)$. Iterating through the length of Clist does not increase this by a factor of $|V|$, since BFS is being performed on subgraphs of the whole graph rather than the whole graph $|V|$ times. The second part of the algorithm is $\mathcal{O}(|V|)$ since the nodes are effectively iterated through again to find the minimum; finding the minimum of several disjoint subsets of a big set is the same order as finding the minimum of a big set. The total cost of this function is therefore $\mathcal{O}(|E| + |V|)$.

Part 2

To make the models easy to analyse I will fix my linear diffusion constant, D, as 1 for this part of the assignment.

2.1.i)

From the definition given in the assignment, for each step of my random walk there is an equal probability of travelling along any neighbour of the nodes, so using *numpy* and its *random* function suffices here. The paths are initially stored as zeros in an Nt by M matrix. The first column is then converted to the starting index, denoting every node has the same starting position. The process I then use is as follows:

I generate $Nt * M$ numbers between 0 and 1. As I traverse my graph (Mth node at time Nt), I multiply the (M, Nt) entry of my randomly generated numbers by the degree of my current node. I then take the floor function of this number (call this x) and select the xth element of the adjacency list corresponding to the current node. This element is then my next step of my path and prevents calling upon the *random* function $M * Nt$ times. My method is approximately 30 times quicker than the naive method which calls upon the random function $Nt * M$ times for when I use $M = Nt = 5000$.

2.1.ii)

$Barabasi2000 = nx.barabasialbertgraph(2000, 4, seed = 1)$

The graph I am dealing with for the remainder of this section uses the above seed.

In my simulation I use Nt and M to be 5000. The idea behind this random walk is that as I increase the values of Nt and M, if I were to produce a histogram of the destination of all the nodes, this would converge to the probability mass function of the system. Large M gives me a more accurate representation of the distribution a particular time.

Thankfully, using knowledge from applied probability, the pmf of the system isn't too hard to derive. At time t we can define the probability of being at node j as $p_j(t)$. This can now be defined recursively as:

$$p_j(t+1) = \sum_i P_{ij} p_i(t)$$

where P_{ij} is equal to $\frac{A_{ij}}{d_i}$ with A being the adjacency matrix and d_i being the degree of node i.

From applied probability, I know that the random walk is reversible owing to the fact that the entire network is undirected. Using this, I know that the stationary distribution of the nodes is the limiting distribution. Using the notion of time reversibility, the following is true:

$$\frac{A_{ij}}{d_i}\pi_i = \frac{A_{ji}}{d_j}\pi_j = constant, \sum_i \pi_i = 1$$

where π_i is stationary distribution of the ith node. The adhacency matrix A is symmetric so this cancels out. Rearranging this equation and using the fact that the sum of the stationary distribution is equal to one, I can deduce the following:

$$\pi_i = \frac{d_i}{\sum_j d_j}$$

The sum of the degrees of all nodes is simply twice the number of edges of the graph, so what can be deduced here is the that the distribution of destinations of random walks should be proportional to the degree of the nodes. Below are two figures. The first figure shows the limiting distribution of the random walk in the form of a pmf. The second figure shows the distribution of destinations of random walks (where all walks start off at the same starting node) in the form of a histogram ($Nt = M = 5000$). What can be seen here is that the distribution vaguely resembles the limiting distribution, with the nodes of highest degree being the most frequent.

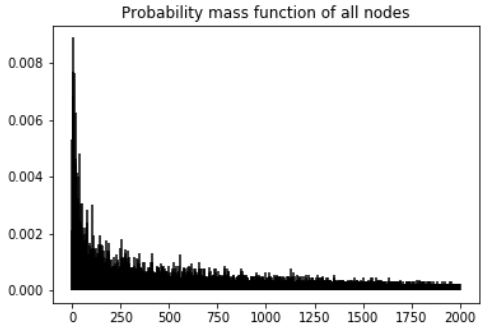

Figure 1: PMF of destinations of nodes after infinite time

The y axis in the first case is the probability of landing on each node as you tend time to infinity, and in the second it is the frequency of destination node, with its density set equal to 1 so that the two graphs are easily visually comparable.

2.1.iii)

In the previous section, the random walk produces a distribution which is weighted according to the degree of each node.

We know that the Laplacian matrix is positive, semi-definite and so all of its eigenvalues are non-negative. As we want to see what the solution is for

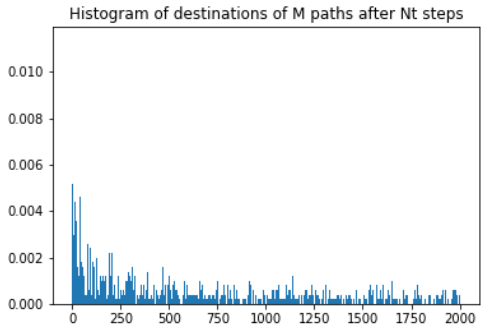

Figure 2: Histogram of nodes for Nt=M=5000

asymptotic time, t, we only wish to find the eigenvalues that equal 0; the eigenvectors corresponding to the zero eigenvalue will tell us about the distribution as time approaches infinity. This is because all eigenvalues other than zero are positive, and the when the exponential is applied to the other eigenvalues, the multiplication by -1 makes the eigenvector corresponding to that eigenvalue vanish as t becomes asymptotically large.

Graph theory tells us that the number of connected components in the graph is the dimension of the nullspace of the Laplacian and the algebraic multiplicity of the 0 eigenvalue. Since this graph is entirely connected, the algebraic multiplicity of the zero eigenvalue is simply 1. Furthermore, the row sums of the Laplacian are equal to *zero*, which means that the candidate vector:

$$\mathbf{v}_0 = (1, 1, \ldots, 1)$$

is in the kernel of our Laplacian matrix. Since we know that the kernel is of dimension 1, we can deduce that as time approaches infinity, the distribution of nodes is uniform. From this, I can deduce that the Laplacian does not produce dynamics akin to the random walk generated in the previous question.

Now I need to consider the scaled Laplacian and its transpose. Figure 3 was produced by using python's *odeint* to give me the quantity at each node at a time,t of $1,000,000$. The initial value of my equation fixed a value of 1 at the source node and 0 at every other node. This was so that when the graph was plotted, the y axis would be directly comparable to the figures produced in the random walk question.

What can be seen from figure 3 is that the transpose of the scaled Laplacian appears to be the only one of the three diffusion operators whose dynamics produce significant similarities to the random walk. The shape of the density is identical to the theoretical pmf of the random walk. As for the scaled Laplacian, the distribution of quantities at each node is uniform, meaning the asymptotic dynamics are equivalent to the non-scaled Laplacian (reason for this given on next page).

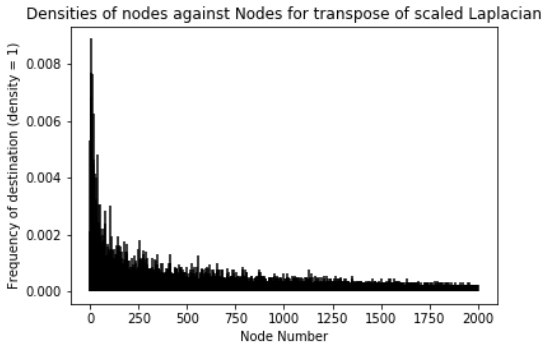

Figure 3: State of nodes of transpose of scaled Laplacian at time, t = 1,000,000

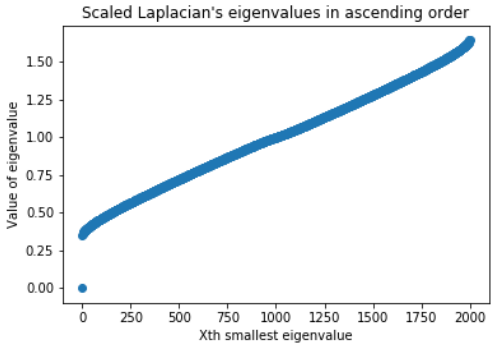

Figure 4: Eigenvalues of the scaled Laplacian Matrix

It is worth stating that the eigenvalues of a matrix and its transpose are equal, so the plot of eigenvalues for the transpose of the scaled transpose is the same. Therefore the dimension of the *kernel* for the Laplacian and its transpose are equal, and from figure 4 we can see that this is *one*. In addition to this, all eigenvalues are positive, so we are again only interested in the eigenvector corresponding to the zero eigenvalue as we t becomes asymptotically large.

What is not the same, however, are the kernels of a matrix and its transpose. For the scaled Laplacian, the following vector is in the kernel:

$$\mathbf{v}_0 = (1, 1, \ldots, 1)$$

This agrees with our finding of a uniform distribution in the limit as t approaches infinity.

For the transpose of the scaled Laplacian:

$$L_s^T = I - AQ^{-1}$$

Considering the multiplication of matrices and the fact that each element of Q is $\frac{1}{d_{ii}}$ along the diagonal and 0 elsewhere, it appears that the following vector is in the kernel:

$$\mathbf{v}_0 = (d_{11}, d_{22}, \ldots, d_{20002000})$$

Since this vector is in the kernel, it makes sense mathematically that the limiting distribution of quantities at each node is proportional to the value of the node, giving us the result identical to that of the random walk in the previous question.

2.2)

2.2.i)

As there are L links, for each link, this corresponds to two entries in our sparse matrix: (i, j) and (j, i). In this instance with our sparse matrix, only the $2L$ elements are considered. Furthermore, since this is vectorised, the multiplication and addition of scalars and our predetermined vector, y, is $\mathcal{O}(1)$. Therefore computing the derivative of i with respect to t takes approximately $2L$ calculations which is far superior to the N^2 calculations that would otherwise be done if a sparse matrix weren't used.

2.2.ii)

$Barabasi100 = nx.barabasialbertgraph(100, 5, seed = 1)$ is the graph with which I am dealing for this section of the assignment.

For this section I constructed model B using *odeint* and making use of the fact that the row sum of a Laplacian is equal to zero. This means that the negative term, s_j disappears when put together with L_{jk}.

In model B, for $\alpha \leq 0$, it appears that the sum of quantities for all nodes is fixed (figure 6). This also happens to be the case for linear diffusion. Therefore, there is a similarity between the two dynamics: the sum of the quantities of all nodes is invariant. This prompted me to discover if there exists an α such that the value of all my nodes is uniformly distributed, since were this to be the case, the two dynamics would behave identically in the limit as t approached infinity.

What can be seen from figure 7 and figure 8 is that while the sum of all nodes is fixed and equal to 1, as α increases, all nodes except for the source node (node 5 with this seed) are approximately the same value and are very close to zero. This appears to be outweighed by a very negative value for the source node. Thus, in the limit as t approaches infinity, it appears unlikely (from the test cases I have carried out) that there exists a negative α such that model B is identical to linear diffusion. Indeed, this makes sense since one of the differential equations closely resembles linear diffusion, though this is counteracted by the time relationship between s and i.

In addition to this, for positive α, the sum of the system appears to oscillate wildly between positive and negative with the oscillations happening earlier on for larger, more positive α.

For model A, I considered four permutations of γ and β. The cases were the pairs where what differs is the sign of the parameters $(+or-)$. What can be seen from figure 5 is that for negative γ, the sum of the total system rapidly approaches negative infinity. For positive values, the graph appears to tend to a constant value with its size dependent on how negative β is.

A similarity between model A and model B is that when given a starting node with a positive value (in this case 1), the system can become net negative, unlike with linear diffusion.

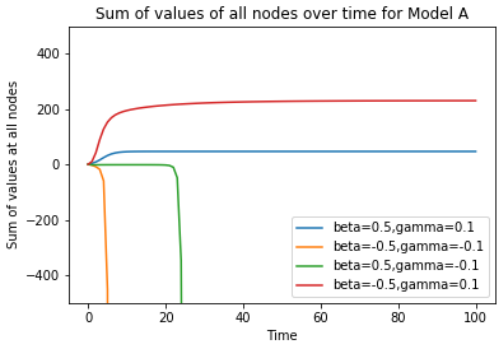

Figure 5: Sum of values of all nodes over time for Model A

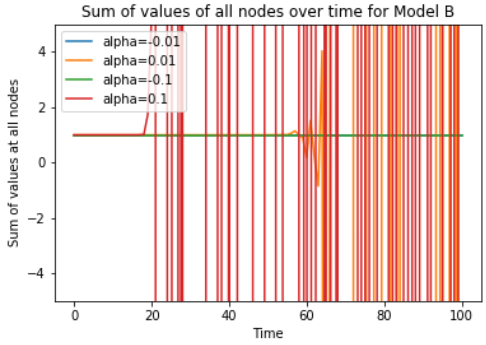

Figure 6: Sum of values of all nodes over time for Model A

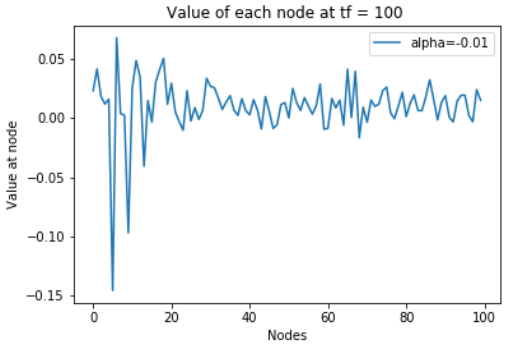

Figure 7: Value of nodes at time t = tf for alpha = -0.01

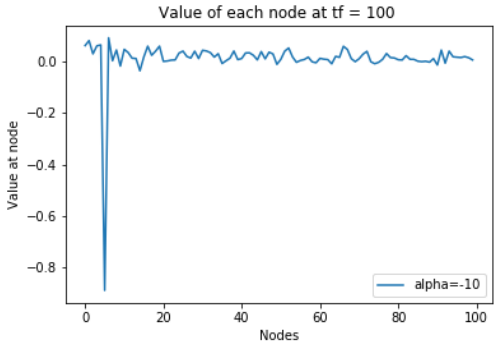

Figure 8: Value of nodes at time t = tf for alpha = -10