Networks Analysis Assignment

Elliot Moffatt

gvch48

Question 1

Degree Distribution

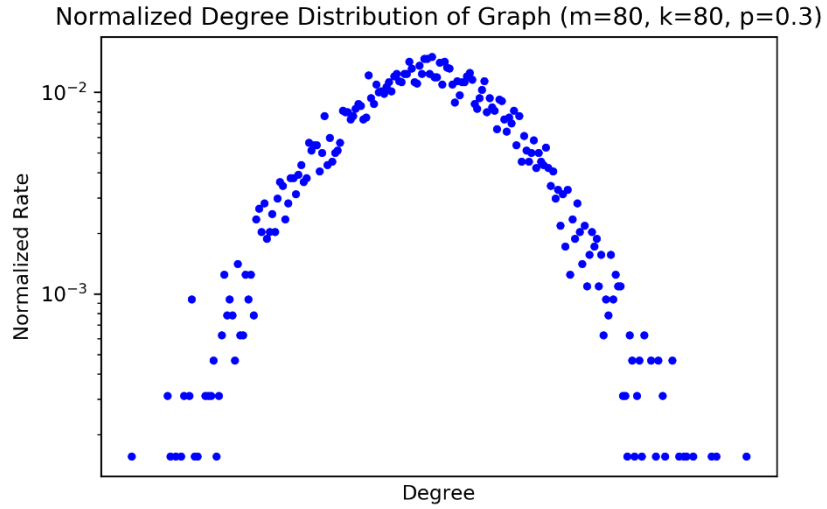


Figure 1: m=80 k=80 p=0.3

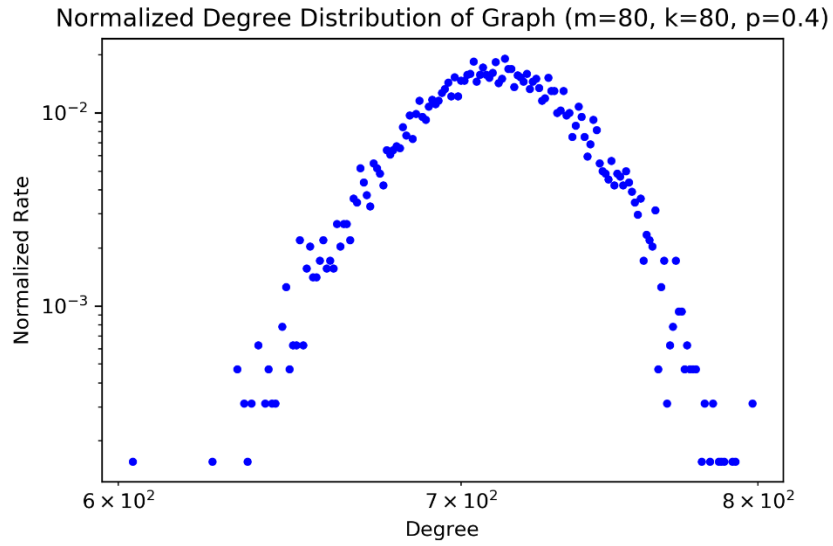


Figure 2: m=80 k=80 p=0.4

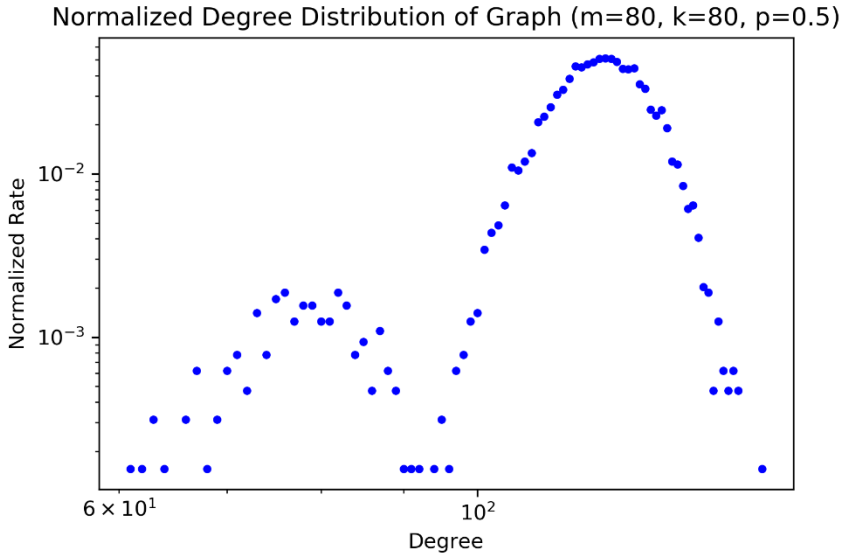
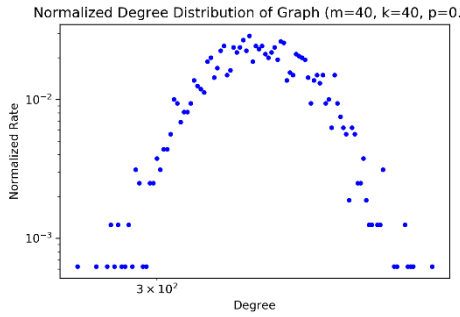
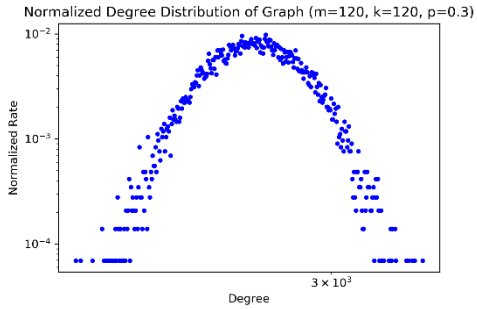
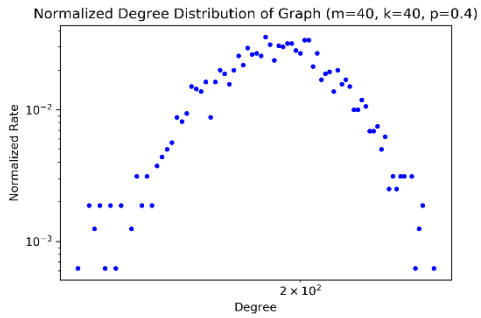
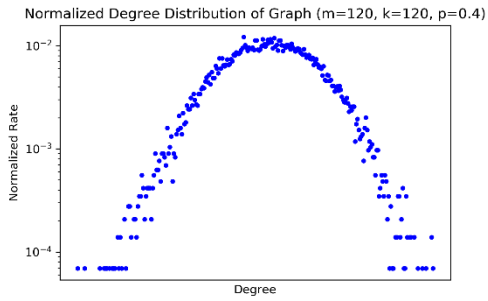
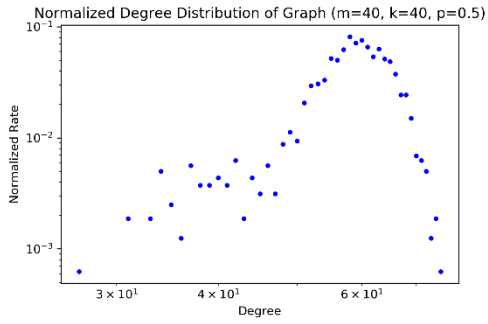
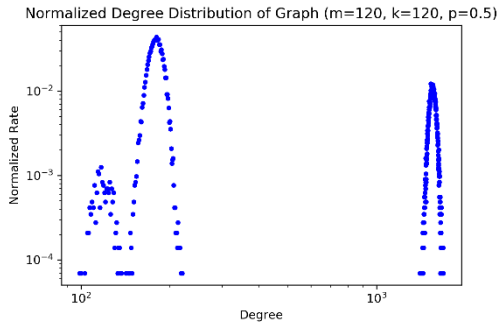


Figure 3: m=80 k=80 p=0.5

When p is similar to q, the degree distribution takes the shape of a bell curve, but as the difference between p and q increases the distribution splits into 2 distinct arches.

I achieved similar results for m=k=40 and m=k=120

	M=k=40	M=k=120
P=0.3		
P=0.4		
P=0.5		

Diameter

For all the graphs I investigated I set the value of q to be very low and the value of m to be much larger than k, so that the value of p has a much larger impact on the diameter. Every plot I made from graphs following these criteria showed that diameter becomes exponentially smaller as p goes from 0 to 1, with the largest change in diameter occurring between p=0 and p=0.2.

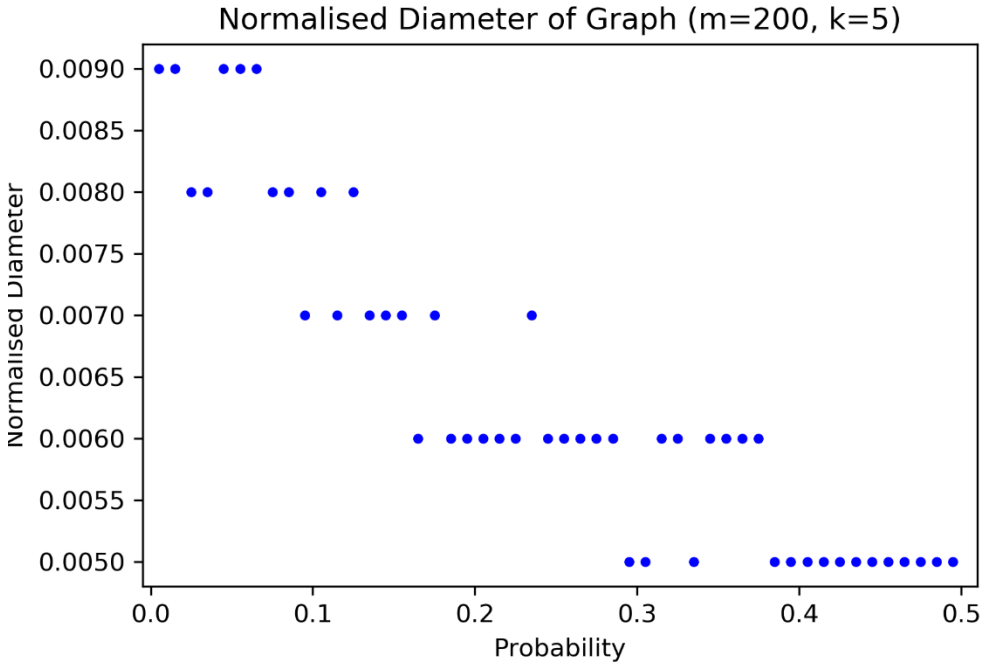


Figure 4: m=200, k=5, q=0.005

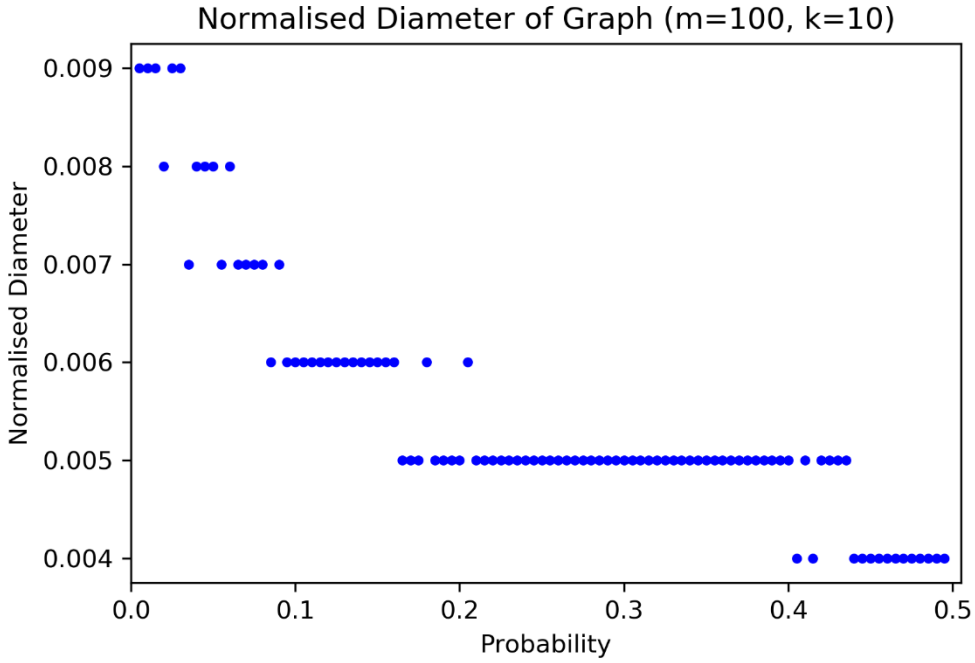


Figure 5: m=100, k=10, q=0.005

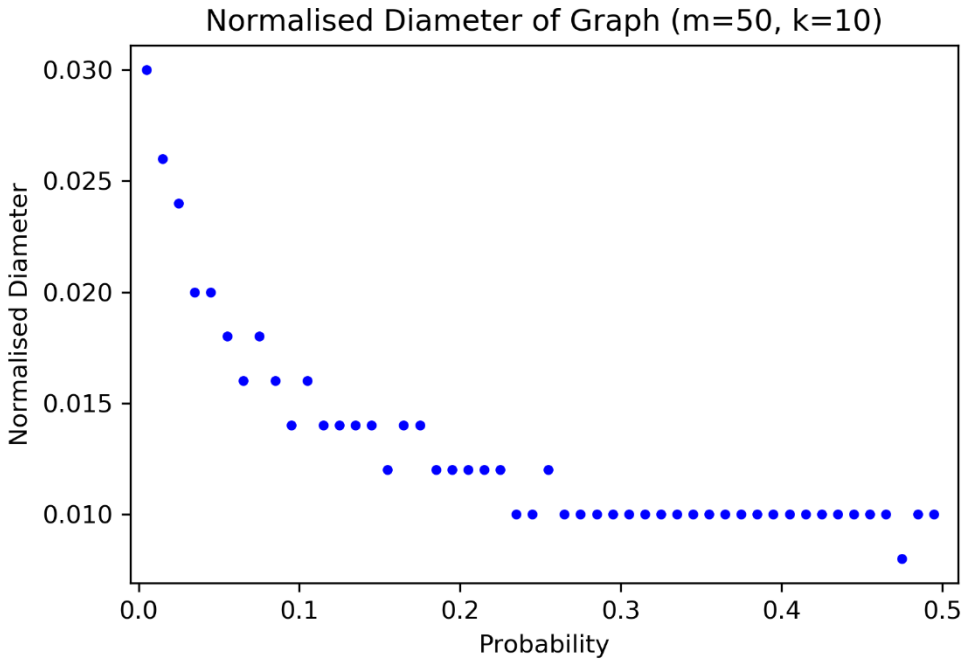


Figure 6: m=50, k=10, q=0.005

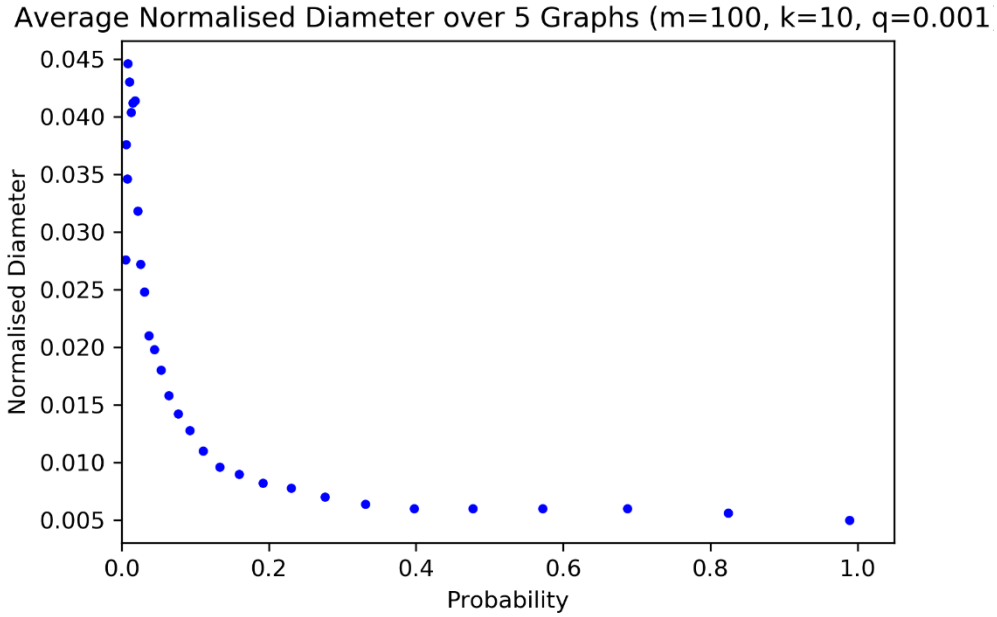


Figure 7: m=100, k=10, q=0.001

Clustering Coefficient

I fixed q=0.2 for all plots made. In general increasing q will decrease clustering coefficient and decreasing q will increase clustering coefficient.

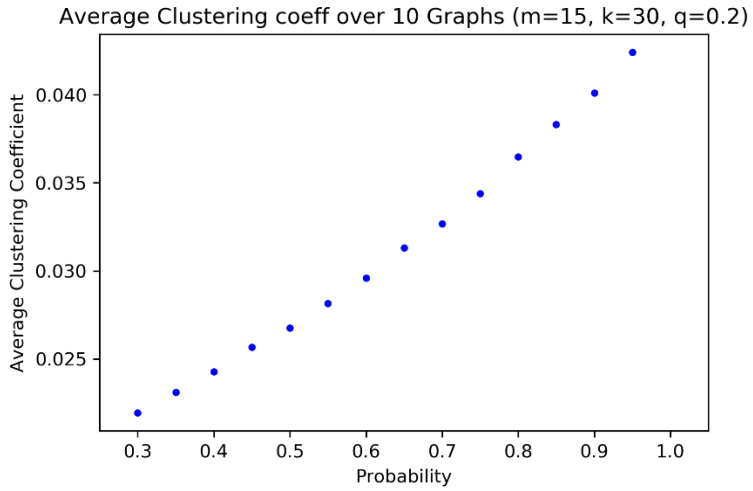


Figure 8

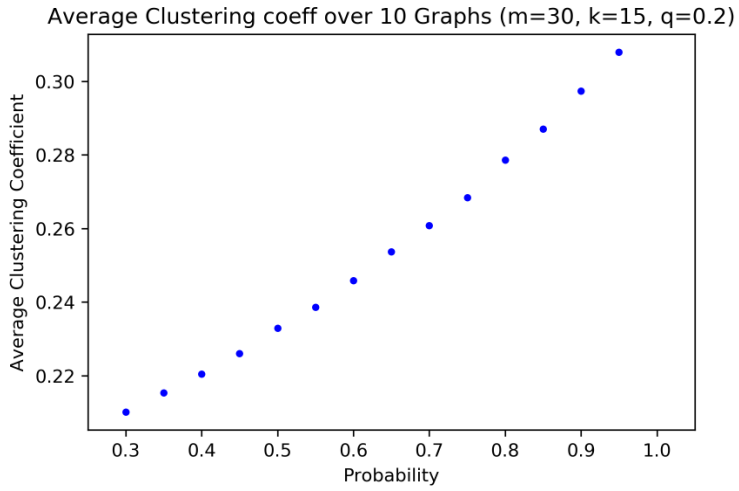


Figure 9

I created plots for m=15, k=30 and m=30, k=15. Both plots look similar, being slightly exponential but effectively linear for the values of p used. The clustering coefficient at each value of p is about 10 times smaller for m=15, k=30 than for m=30,k=15. This is expected because for the first there are many more vertices not in the same group and given q is somewhat large each vertex will have many more neighbours not in the same group.

Question 2

Brilliance Function

I created two different functions to calculate the brilliance of a vertex

Method 1: The first started with all neighbours in the “star”, and iteratively deleted neighbours that were connected to the most other neighbours, until none of the neighbours were connected, at which point the star actually is a star and the size of the star is returned.

Method 2: The second started with an empty star and added all the neighbours to a selection pool. The function finds the neighbour in the selection pool with the fewest other neighbours also in the selection pool and adds it to the star, then removes it and its neighbours from the selection pool. It repeats this until the selection pool is empty, then returns the size of the star.

Coauthorship graph

I computed the Normalised Brilliance Distribution for this graph using both Brilliance functions above. Interestingly, while both graphs are very similar, there are slight differences, which I did not expect since the two methods should produce the same results.

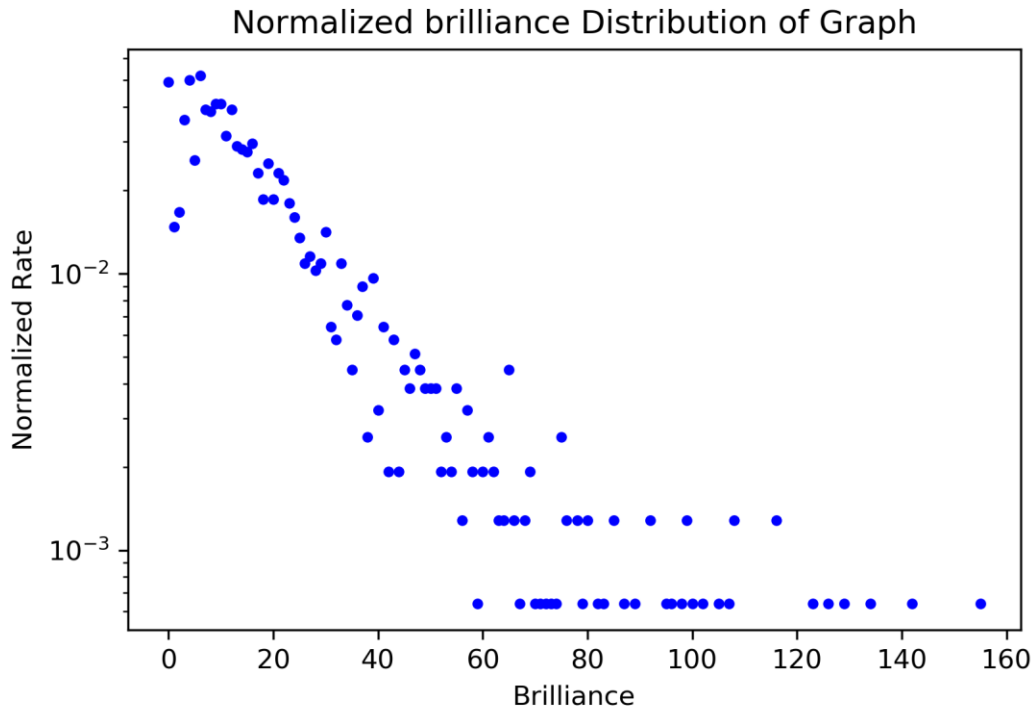


Figure 10: coauthorship graph using method 1

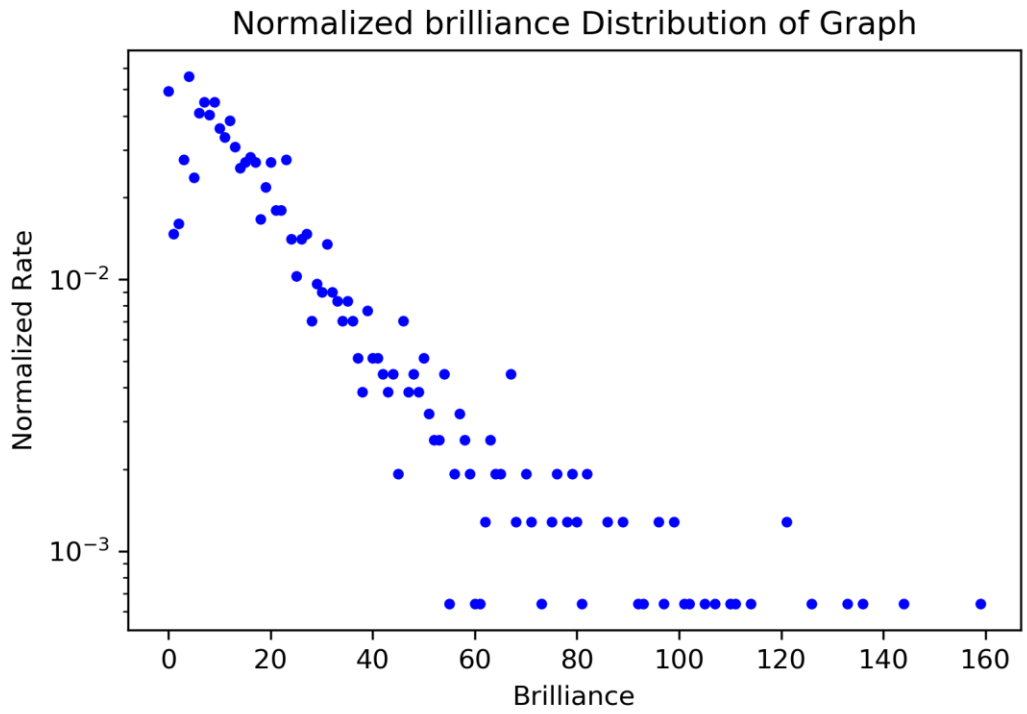


Figure 11: coauthorship graph using method 2

As expected, the majority of authors have a very low brilliance. However, I was surprised to see authors having brilliances of more than 140

Ring Group Graphs

The Coauthorship graph has 1559 vertices and 40016 Edges. To produce a Ring Group graph with similar values for vertices, V, and edges, E, as the coauthorship graph, I fixed m=100 and k=16 to get a similar number of vertices. The Expected number of edges, E, of a ring group graph is

$$E = m * k * ((3k - 1) * p + (m - 3) * k * q)$$

By choosing p=0.2, we find that q=0.010058 gives us a similar number of edges to the coauthorship graph. The result of normalized brilliance for a ring group graph with these parameters is shown below.

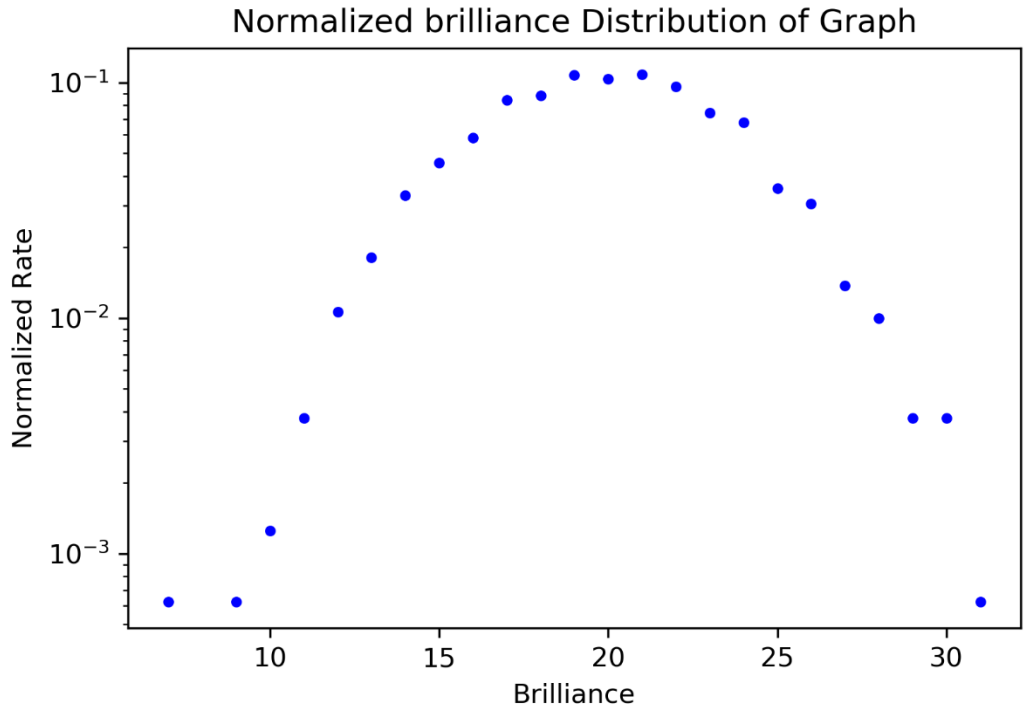


Figure 12: ring group graph, m=100, k=16, p=0.2, q=0.010058 using method 2

The symmetric shape of the graph (ignoring the anomalies created by only looking at one graph and not averaging over several) is expected since each vertex in a ring group graph is constructed the same way with the same probabilities of connecting edges. The most interesting things to note is that the range of brilliances, as well as the average brilliance, is very low compared to those of the coauthorship graph. I expect that by varying values of m,k,p and q (while keeping V and E constant) could lead to much larger brilliance values, although the shape of the plot would remain the same.

PA Graphs

Similarly to Ring Group Graphs, I calculated values of total vertices (m) and out degree (n) that kept V and E constant and similar to those of the coauthorship graph. Trivially, m = V (=1559)

Then $E = n(n-1) + (m-n)n$

Simplifying: $E = n(m-1)$

And we get out degree n=26. The plot for a graph with these parameters is shown below.

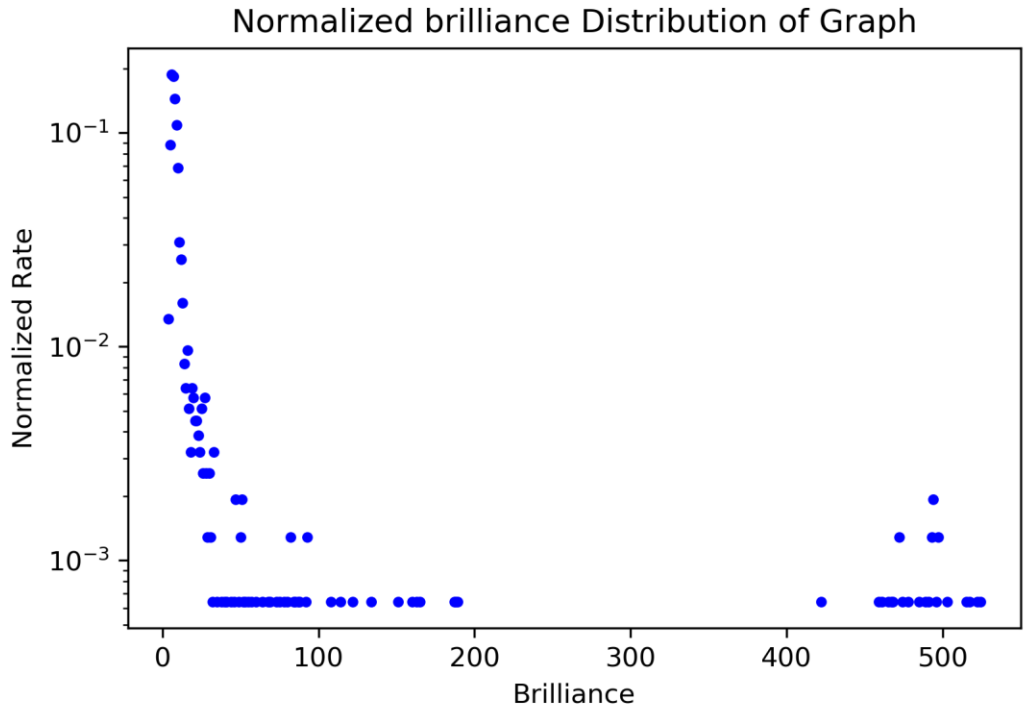


Figure 13: PA graph, v=1559, n=26, using method 2

This plot is much more similar to the coauthorship plot than the ring group plot. Most brilliances are close to zero but the max brilliances are very large. This is expected since the first constructed vertices are likely to (because the graph is undirected) be connected to a lot of vertices that were constructed much later, and since these later constructed vertices are very unlikely to be connected, it makes the early vertices the center of very large k-stars.

Question 3

For each type of graph, I wrote an algorithm that computes the search time for a given source and target vertex. Whenever I reference an algorithm below, I am referring to the algorithm for finding the search time of a given pair of vertices. To calculate the search time of the graph as a whole I wrote another function that samples this algorithm over a given number of sample pairs of vertices (chosen randomly but checking that the pair aren't the same vertex chosen twice and that the pair hasn't been sampled before)

Random Graphs

Since every vertex has the same probability of being a neighbour with the target vertex, there is no advantage to moving to a neighbour before checking all the vertex's neighbours. Thus, the algorithm queries every neighbour and if none of them are the target then it picks a neighbour at random to move to next.

I set this function to run for a random graph with 1000 vertices, edge probability 0.05 (so each vertex will on average have 50 Neighbours, and the graph will have on average 50000 edges) I then sampled the graph 1000 times and took the average search time. I plotted the search time of 3000 such graphs on figure 14, with the average search time being around 1025.

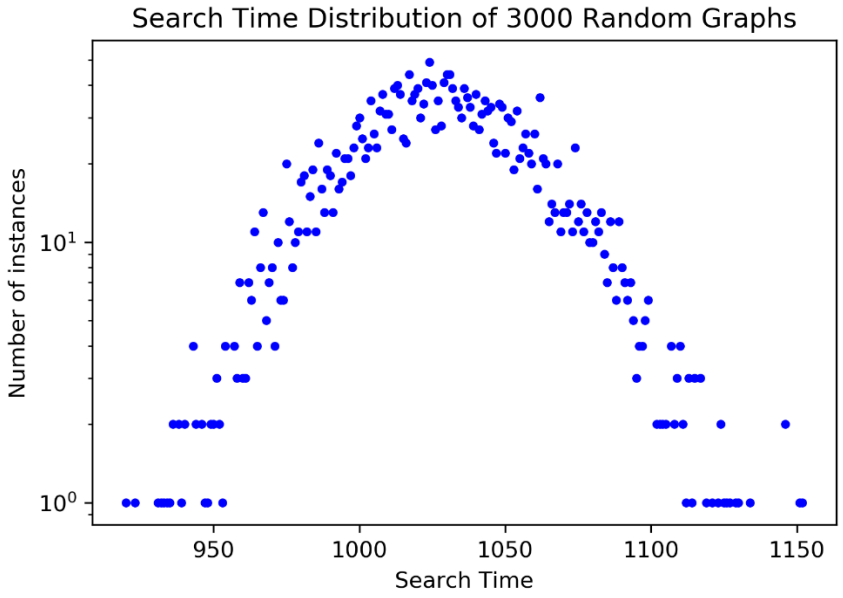


Figure 14: Random graphs, V=1000, p=0.05

Ring Group Graphs

The algorithm assumes that p is (much) greater than q. It first checks if the current vertex is in the same or adjacent group to the target vertex. If it is then there's a very high chance that one of its neighbours is the target so it checks every neighbour before moving. If none of its neighbours are the target then it calls the choose_best_next_neighbour function, which I shall explain later.

If the current vertex isn't in the same or adjacent group to the target, then it begins querying its neighbours. After querying each neighbour, it checks how close the neighbours group is to the targets group (A closer group is better because in the "worst case" scenario where q=0 the only way to get to the target is by moving between adjacent groups) The algorithm then calculates the odds of one of the unchecked neighbours being closer to the target than the currently considered neighbour. If the odds of finding a better (closer to target) neighbour are less than 50% then the algorithm jumps to the neighbour straight away instead of querying the rest of the neighbours. If the algorithm checks all neighbours (i.e. hasn't jumped early) then it calls choose_best_next_neighbour().

Choose_best_next_neighbour() is given the number of groups, the list of neighbourIDs, and the targetID. It sorts the list of neighbourIDs based on distance to target, with neighbours with smaller distances at the front. It then decides which neighbour to jump to. It jumps to the first neighbour with probability 0.5, second with probability 0.5^2, third with probability 0.5^3 etc. While this technique reduces the average

efficiency of the algorithm by not always picking the optimal choice, it prevents the worst-case scenario of becoming deadlocked by jumping between the same 2 vertices because they are each other's optimal choice.

I ran this algorithm over 3000 graphs, sampling 1000 vertices per graph, where each graph had the values m=100, k=16, p=0.5, q=0.01 (1600 vertices, around 62432 edges). The average search time was around 238, with a few anomalies with much higher search times.

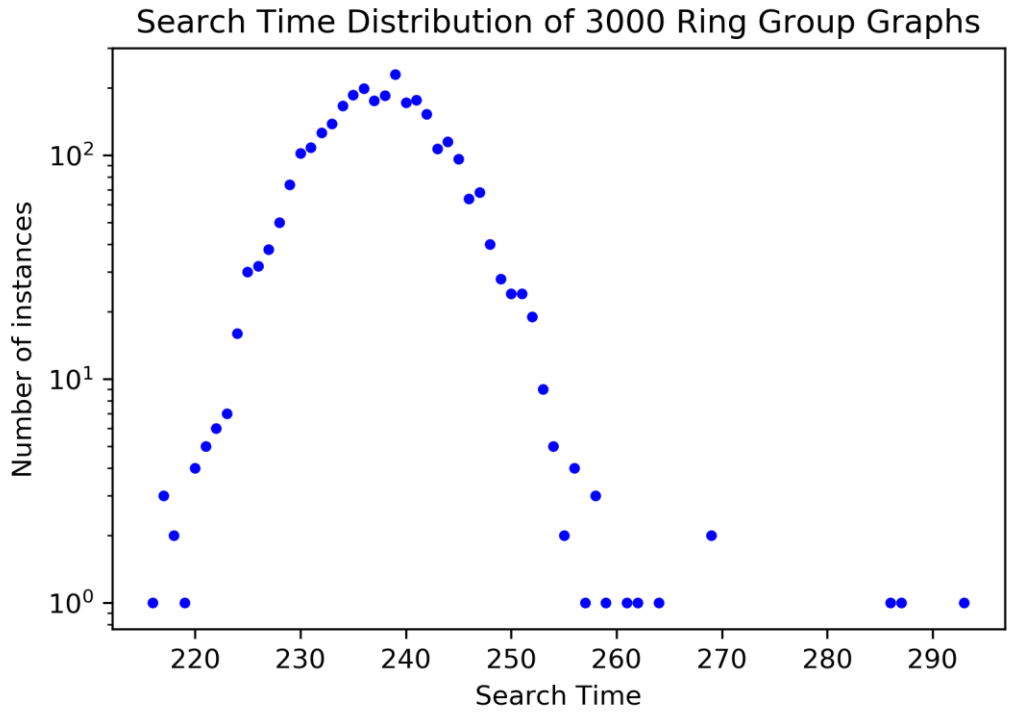


Figure 15: Ring Group Graphs, m=100, k=16, p=0.5, q=0.01

PA Graphs

The algorithm is built from the concept that since the PA graph is undirected, the m vertices from the original complete graph are very likely to be connected to the target.

The algorithm checks if the current vertex is from the original complete graph. If it is then it queries each of its neighbours to see if they are the target. If none of them are then it moves to one of the other m-1 vertices from the original graph and repeats the process

If the current vertex is not from the original complete graph, it queries each of its neighbours. When it finds one that is from the original complete graph it jumps to it straight away. If it doesn't have a neighbour from the complete graph, it sorts its list of neighbours and randomly jumps to one of the 5 neighbours with the lowest ID values. Again, this randomness is to prevent deadlocking and due to the nature of PA graphs the lowest 5 neighbours are all likely to have very low IDs and will likely share most of their neighbours, so this randomness shouldn't affect the search time much (if at all). Of course, this technique assumes the out degree is at least 5. The algorithm can easily be adapted to adjust or remove the randomness described above to account for lower out degrees, but since I decided to run this algorithm on large PA graphs with large out degrees I decided that adapting the algorithm wasn't necessary.

I ran this algorithm over 3000 PA graphs, sampling 2500 pairs of vertices per graph. Each graph had 1000 vertices and an out degree of 20 (so average number of edges is 19980). The average search time was around 745

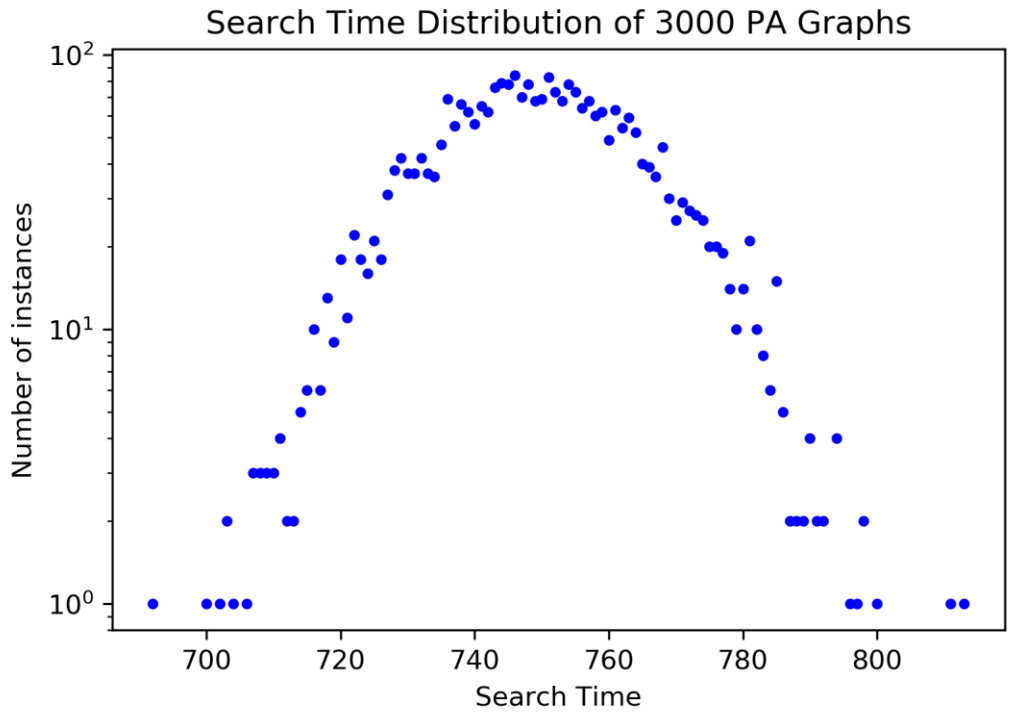


Figure 16: PA graphs: V=1000, out degree=20

Comparison of Results

Due to the different sizes of the graphs I ran the algorithm on, comparing the raw search times does not provide a lot of insight into the efficiency of the algorithms. Comparing search time / edges gives a more useful (but still not perfect) insight, since average number of edges is dependent on all of the graphs parameters for all three graph types.

Graph Type	Random graph	Ring Group graph	PA graph
Average Search Time	1025	238	745
Average Edges	50000	62432	19980
Search Time / Edges	0.0205	0.00381	0.0373

Figure 17

By comparing search time / edges, we see that the algorithm for the ring group graph performs best, about 5 times better than the algorithm for random graphs, which is about twice as good as the one for PA graphs. However, the difference in values is still relatively small and given how imprecise this metric is at measuring the success of the algorithm I would not make any conclusions without further research.