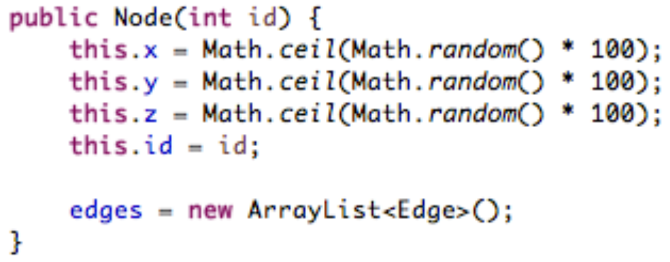
Ryan Bergquist
Math 466
Final Project

1. **Final Project Summary**
   a. **Random Networks**
      i. We were to generate random networks given n nodes, and m arcs. Each of these networks was to be designed using integer capacities that have been randomly chosen from the interval [1,U]. This can be easily done in a few different ways. In the main class file, the entire program is set up to run a loop ten times, the number of different networks required to analyze. To do this, a few things must be done on each pass.
      ii. To start, the user must start with a few variables, the number of nodes, and number of edges, source, and sink. These can be manually chosen, but since the assigned calls for random networks, using Java's built in random functions seems efficient. Java's built in Math.Random() function was used in designing all parts of the network. By calling Math.Random() we get a random number between 0 and 1. Since we are looking for integers since we need number of nodes and edges, and there is no such thing as half and edge or node, then we must multiply the double that is returned by a multiple of 10 get a number between 1 and 100, depending on what we want. Then taking just the ceiling of the number, we have an integer that we can use to generate new graphs.
   b. **Generate Graphs**
      i. For each pass, we generate new graphs given an number of nodes required, the number of edges that exist within the graph, and the maximum degree that each node may have. The maximum degree is used to limit the edges making the graph more realistic. To start, the graph holds on to an array list of the nodes and edges in the graph. There is also an id stored for each graph so they may be stored, and easily located for later use.
      ii. The graph class starts with adding new nodes to ta list of existing nodes that the graph keeps. The way the nodes are made will be discussed later. After the nodes have been created, the graph class runs the list of nodes, and connects them to a random other node. After the list has been traversed, if not enough edges have been created, then the list is traversed again until complete. Through each pass, it is necessary that we check that no node has a degree higher that the limit that has been set. If a node is full, then we have a problem, and we must make a new graph. If this happens, we simply return a value that signals and error has occurred, so we get new random numbers, and make a new graph.
      iii. When connect to a random node, we can once again take either the ceiling or by casting as an integer, a random number that is multiplied by one less than the size of our node list. If the node trying to be connected to is full, it loops round to all other nodes to try and connect to it.
   c. **Generate Nodes/Edges**
      i. When generating new nodes, we start with our graph, where only a empty list of nodes and edges have been created. We count from 0 to the number of nodes that we want, and add to our list, new generated nodes with an id assigned from 1 to the number of nodes.
      ii. Since only the id number for each node is passed in, we must find a way to generate new nodes. To do this, we can picture a network being drawn on paper, and each node is placed on a different part of the page, so they consist of an x and y co-ordinate. To ensure that distances from each node or more distinct numbers, we can add a z co-ordinate when

assigning values. As we have continued to do, we will just Javas Math.Random() function again to generate different x, y, and z values. Using the following function, we compute integer values for the x, y, and z values for all nodes needed.



```
public Node(int id) {
    this.x = Math.ceil(Math.random() * 100);
    this.y = Math.ceil(Math.random() * 100);
    this.z = Math.ceil(Math.random() * 100);
    this.id = id;

    edges = new ArrayList<Edge>();
}
```

iii. Once a list of nodes is created, and the basis of the graph is set up, we must follow further down or graph class to where we begin to connect these nodes. Before an edge is created, a few asserts are necessary. We start by calling a function given a node A. While picking a random node from the list, we check if the nodes are already connected, or if they are the same. If both have been passed, we must check if we are at the end of the node list, or if the nodes have to high of degree. As long as these remain true, we can connect the nodes.

d. 10 Different Graphs

i. As these graph are created, a few restrictions have been assigned. Due to readability, were do not require a large amount of nodes and edges to show how the algorithms work. So when creating the graphs, as mentioned above, a multiple of 10 is used to determine the number needed.

ii. We start with the upper bound, U. This has been capped at 100, since we have a good chance of having a upper-bound that is not always the same since we are generating only a few random numbers for each graph.

iii. On average, the graphs used in example range between 4 and 8 nodes. Since I am testing the run-times using computing methods, the range can be increased. Therefor we can compute random numbers and place a check that the number is between 5 and 20 nodes. Since we relying on a higher amount of nodes, then we can get an edge count between 20 and 40.

e. Run-Time

i. On each of the following Maximum Flow algorithms, we are to measure the run time of each of the algorithms implemented, and how they compare to the theoretical run-time. We must also compare the run-time to each of the other algorithms. So for each our graphs, a few variables can be placed in order to store the start and end times of each algorithm for later comparison.

ii. To get a time stamp, just before the new algorithm has been called, we call System.nanoTime() function. Just after the algorithm finishes, and before the end result is printed, the same time function is called to get the end time in Nano-seconds. These times are not from the clock, but from a set reference point. Since they use the same reference, we can take the difference of the two in order to compute the total run-time in microseconds. Using the wrong time function may not give a small enough number to notice any difference.

2. Shortest Augmenting Path

a. Theoretically, the Shortest Augmented Path algorithm is assumed to run in **O**(n^2m) time. We start with an initial flow of 0, and are given the graph represented as an int[][], the number of nodes (easily taken my int.length), the source, and destination.

b. The algorithm works in a single look, where as long as a path exist through a breadth-first search, we augment a path. We start with initializing the prev array to all hold the value -1. We start by adding our source to our queue and loop. If our temporary arrow of the flows is greater than 0, or the value from our current node is less than the capacity of the current graph, we make marks to show what nodes create a path to our destination, looking at nodes that we have not seen yet.
   i. The total runtime of the BFS algorithm is dependent on the number of verities and edges. We can represent the run-time at O(|V| + |E|) which is reduced to O(|V^2|). Since we are representing number of nodes at n, the current run time is O(|n^2|).

c. Next we must find the minimum value along the path, where we must check m edges on n nodes. Since we are using the prev array mentioned earlier, we know the path from s to t. This allows the runtime to be reduced to linear time. We fun through two separate, un-nested for loops to compute the bottleneck and update our flow network. This allows the second part of our algorithm to run in O(m) runtime.

d. Since no more seconds are to the algorithm we end with runtime O(|n^2|) + O(m) resulting in O(|n^2m|).
   i. We can use this and take our randomly generated values of n and m and compare how long the code took to how it compares to the theoretical values.

3. **Preflow Push Relabel Algorithm**
   a. The FIFO preflow push algorithm is known to have longer run times at O(n^3). We start the same as before, given a graph and flow matrix, source, and destination. We start by getting or number of nodes from the command graph.length if not given.
   b. To start, we set up a few arrays from storing results, excess, height, list, and seen. After setting up a list, we call push through a loop were we send the minimum between capacities – flow and the excess on the edge.
   c. Using the same list, we pull values for u to find the old height. Using the excess array, height, and seen, combined with our current value of u, we can then call the discharge method.
      i. Discharging allows excess flow to be pushed. We must check if the capacity – flow is greater than zero. While we sort through our list, we also maintain a list of already seen nodes. While excess does exist, if at any time we check if using our seen array, if seen[v] is less than the number of nodes, we relabel our graph.
   d. One finished discharging; we can run through our flow array, adding how much is coming from the source to all other nodes, adding the result to a single value. This end value is our maximum flow.

4. **Capacity Scaling Algorithm**
   a. The capacity scaling algorithm is supposed to run in a shorter amount of time, putting its worst case at O(mn LOG(U)). This is the only one of the algorithms where to time complexity seems dependent on the maximum amount of flow allowed through a network.
   b. To start, we find the maximum flow of the graph, assuming it is not given, and set an initial value of delta using 2 to the power of log(max capacity)/log(2). This leaves a cap on how must to augment per cycle. What delta remains larger than 1, we continue the algorithm.
   c. The rest of the capacity algorithm is designed very similar to Edmonds Karps maximum flow algorithm. During the loop, if we have a path from our source to destination, we augment the minimum amount of flow from s to t.
   d. Although I feel I was close, I was unable to fully implement the capacity scaling algorithm so the rest of the paper describing the observed times will be covering the first two algorithms.

5. Plot observed running times

a. The following is a table showing the values after each run of the program

Graph	Num-nodes	Num-edges	Push-time	Expected	Aug-time	expected
1	15	18	13221	5832	1360	4860
2	11	22	8615	10648	2012	5342
3	13	34	45069	39304	31992	15028
4	12	26	71610	17576	5718	8112
5	14	25	12466	15625	14141	8750
6	14	26	13077	17576	19194	9464
7	14	30	22500	27000	17624	12600
8	13	34	45069	39304	31992	15028
9	15	31	49116	29791	1743	14415
10	12	29	38358	24389	23649	10092

b. With the above table, we can see that the expected run time is open expected to be shorter while running our augmented path algorithm. On average, the expected run time is cut in half by not using the FIFO preflow push route.

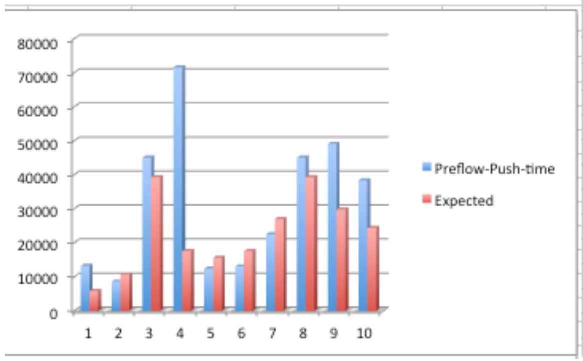
6. Compare to average running time

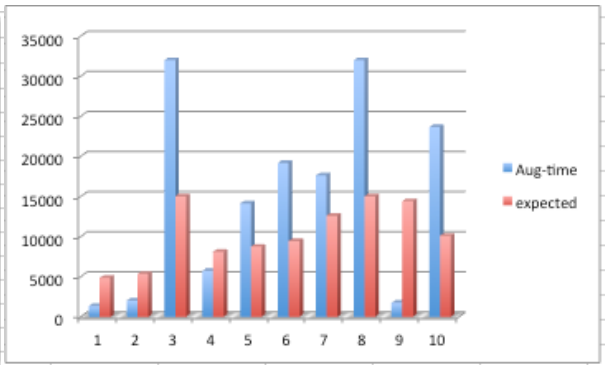
a. The average runtime while running the implement versions often appears to be much higher. Our actual run-time is nearly twice the about of time expected for each run. Some of these could be due to other factors.

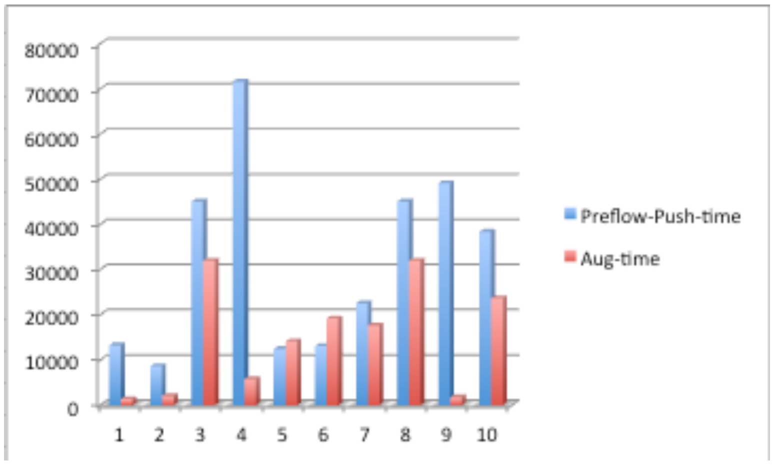
b. The runtime is expected assuming that is the only computation going on. As the CPU heats up, it is easy to see that the run-time tends to increase.

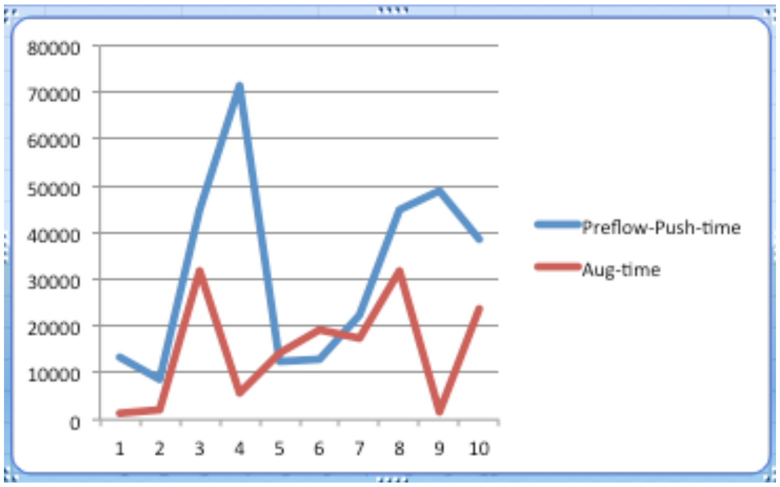
   i. Without having our capacity scaling algorithm which I assume to run in less time, it is hard to tell how the average run time compares across all methods.

c. The following are charts displaying comparisons between the two









7. **What would is the preferred?**
   a. From the methods implemented above, it appears that the shortest augmented path is the best option. It is expected to run in a lower amount of time. Since we know that the expected amount of time is lower, and from our test runs, the implemented version also runs faster, it is clearly a better option in our situation.
   b. On average, the FIFO preflow push algorithm run time was much larger than what it should have been. This makes me believe that I either am not computing the average run time accurately, or the implementation is not correct.
   c. As of now, the two algorithms produce the same maximum flow across the network, so implementation appears to be okay, but run-time does not agree.