CSCI 3104 Spring 2017 **Problem Set 9**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)

1. (60 pts) Recall that a simple breadth-first search algorithm can be used to solve the All Pairs Shortest Paths (APSP) problem for a simple graph $G = (V, E)$. When G is given in an adjacency list format, this takes both $O(V + E)$ time and space. Across all pairs $i, j \in V$, the shortest path with the longest length is called the *diameter*, and the average length across all such paths is called the *mean geodesic distance*.

   In this problem, we will implement and apply four functions.

   (i) `computeDistances(G,s)` takes as input a simple graph $G = (V, E)$ in an adjacency list format and a node $s \in V$, and it returns a vector containing the length of the shortest paths from s to all nodes $t \in V - s$ *and* a tuple containing the number of atomic operations and the maximum length of the queue it took to compute this vector. Nodes that are unreachable from s should be given a special infinite distance value.

   (ii) `largestComponentSize(G)` takes as input a simple graph $G = (V, E)$ in an adjacency list format and returns the size of the largest component in G as an integer.

   (iii) `howBigIsThisGraph(G)` takes as input a simple graph $G = (V, E)$ in an adjacency list format and returns two tuples, one containing the diameter and the mean geodesic distance of G, and one containing the number of atomic operations and the maximum length of the queue it took to compute the first tuple. This function should call `computeDistances(G,s)`. When you calculate the diameter and mean geodesic distance, drop any terms that represent an infinite distance.

   (iv) `erdosRenyiGraph(n,c)` takes as input two positive integers n, the number of nodes in the graph, and c, the average degree of a node, and returns a simple graph $G = (V, E)$, in adjacency list format, such that $|V| = n$ and $\forall_{i>j}\,(i, j) \in E$ with probability $c/(n - 1)$. (Don't forget to add (j, i) to E to make the graph undirected.) This kind of a graph is called an Erdös-Rényi random graph.

   (a) From scratch, implement the functions `computeDistances`, `largestComponentSize`, `howBigIsThisGraph`, and `erdosRenyiGraph`. You may not use any library functions that make their implementation trivial.

   Submit (i) a brief explanation for each function that explains how you implemented it (describe how it works and how it uses its data structures), and (ii)

your code implementation, with code comments.

Note: The code for the functions are located at the very end of this document.

computeDistances(G, s): In order to find the shortest paths from a source vertex s to all nodes $t \in V - s$, we use a breadth-first search on the graph G. Breadth-first search discovers all reachable vertices in G layer by layer, and the distance from s to a newly discovered vertex is always a shortest path, since G is an unweighted simple graph and we don't have to worry about weighted edges. We use a queue data structure to visit the vertices, an array called `visited` to keep track of all visited vertices, and another array called `distances` to store the distance from s to each vertex. If a vertex is *not* reachable, the distance to that vertex is stored as $+\infty$. When the breadth-first search is finished, the function returns the `distances` array, the maximum length of the queue it took to compute all the distances, and the number of atomic operations.

largestComponentSize(G): The `largestComponentSize` function computes the largest component of the graph G, meaning the largest number of vertices that are connected to each other if G is not connected. If G *is* connected, the function simply returns the number of vertices in G. How the function works is that it goes through each vertex u in the adjacency list of G and does a breadth-first search starting from u, and then assigns a component number to the tree discovered by the breadth-first search. The component number for each vertex is stored in an array called `components`. After each u is visited, the function counts the number of vertices in each component and stores component sizes in another array, `numcomponent`. Finally, the function returns the largest value in `numcomponent`, which is the size of the largest component of G.

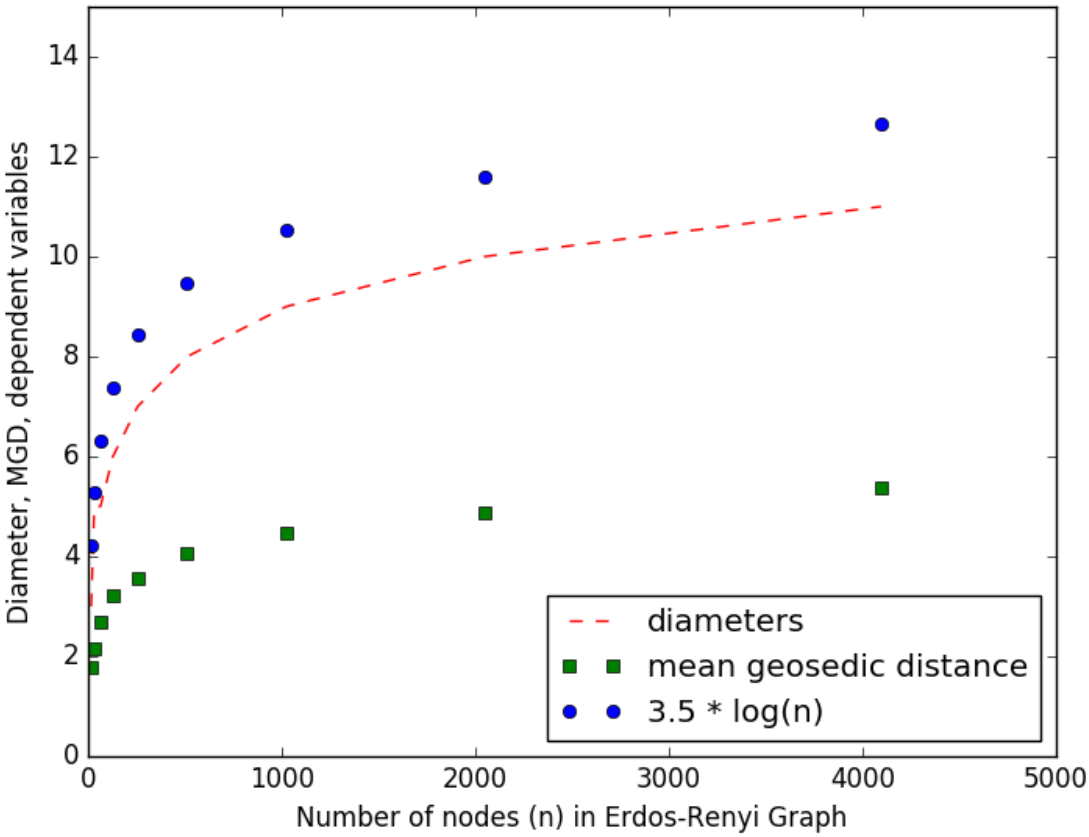
howBigIsThisGraph(G): This function computes the diameter and the mean geodesic distance of a graph G. It first visits each vertex u in the adjacency matrix of G and finds the shortest paths to every other vertex $v \in V - u$ by calling the function `computeDistances(G, u)`. This function returns the array `distances`, which stores the distance from u to each v. It also returns the maximum number of items in the queue that was used to compute all of the distances. Then, `howBigIsThisGraph(G)` goes through the `distances` array of each u and compares every single distance to the current value of the diameter of the graph. If it is greater than the current diameter, then the variable `diameter` gets updated.

Also, the function updates `sumpaths`, which is the sum of all of the distances encountered so far in G, and `numpaths`, which is the total number of paths so far. When the function finishes going through each u, it returns the `diameter`, `sumpaths/numpaths`, which is equivalent to the mean geodesic distance, the maximum length of the queue, and the number of atomic operations.

ErdosRenyiGraph(n, c): This function generates a random graph using the *Erdos-Renyi* method. The function first initializes an array of arrays that will be the adjacency list representation of the generated graph G (each separate array being a single adjacency list). Then, for each possible edge $e \in E$ that could occur in the array given n number of total vertices, the function has a way of figuring out whether that edge will be part of the graph or not. Each edge has a probability of $c/(n-1)$, where c is the average degree of a vertex. Given this fact, the function generates a random number called `edge` such that $1 \leq$ `edge` $\leq n-1$, and checks whether `edge` $\leq c$. Since any number that is less than equal to c will have a probability of $c/(n-1)$ of being chosen, this is a good way of determining if an edge should be in the graph or not. If the function decides that an edge (i, j) will be in the graph, than we add this edge to the adjacency list, as well as the edge (j, i), since our graph will be undirected. Once the graph has been generated, the function returns the graph.

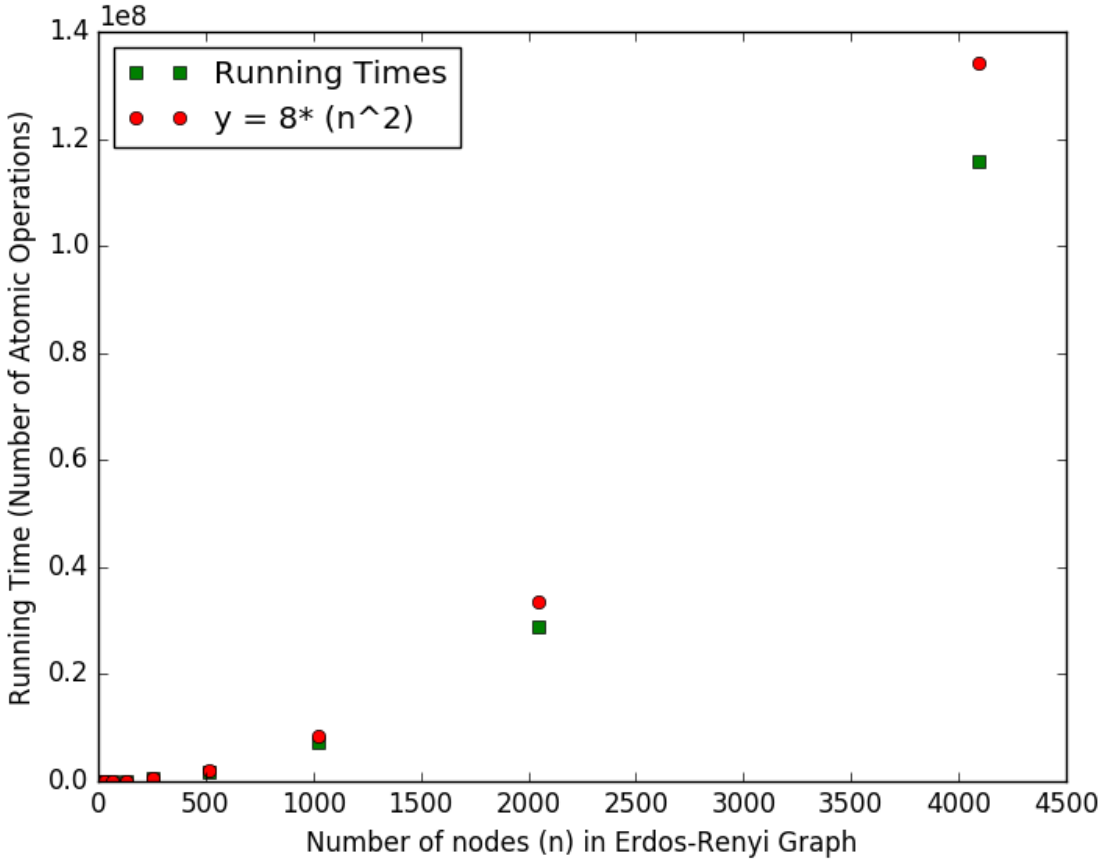
(b) Letting $c = 5$ and $n = 2^k$ for all integers $4 \leq k \leq 12$ for the function call `howBigIsThisGraph( erdosRenyiGraph(n,c) )`, produce a single nice figure showing how both the *diameter* and the *mean geodesic distance* vary as a function of n. Include on your figure a line showing the asymptotic behavior of the way these quantities vary with n. State the asymptotic relationship. No credit if axes are unlabeled or if the three lines on the figure are unlabeled.

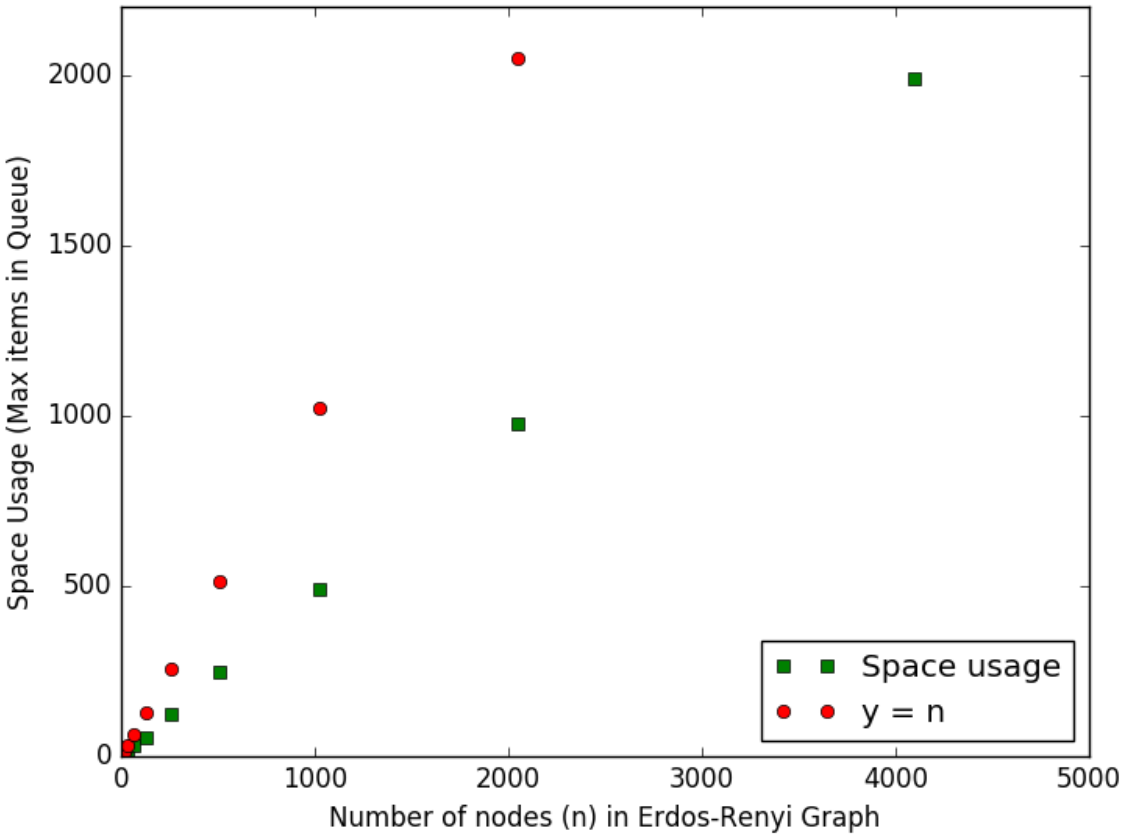
Hint: you will get a smoother line if you run the code several times for each k and then instead plot the average value of the diameter or mean geodesic distance.



The asymptotic relationship for both diameters and mean geodesic distance is $O(\log n)$.

(c) For the same choices of c and k above, produce two nice figures, one showing the running time and and one showing the space usage. Include on your figure a line showing the asymptotic behavior of the resource usage, and comment on whether it agrees with our analysis in lecture. No credit if axes or lines on the figure are unlabeled.


1e8
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Running Times
y = 8* (n^2)
Running Time (Number of Atomic Operations)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
Number of nodes (n) in Erdos-Renyi Graph

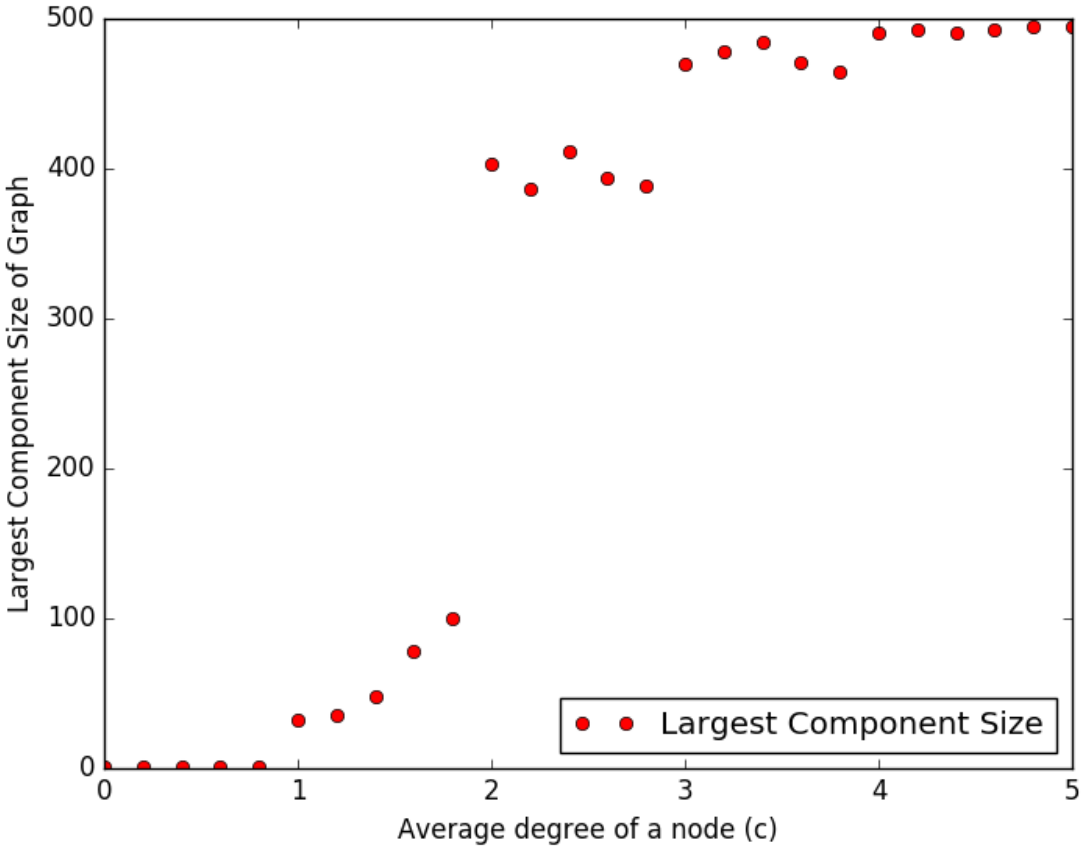


The two graphs above agree with the analysis that we did in class. Our implementation looks very similar to the Bellman-Ford algorithm which has a worst-case running time of $O(VE) = O(V^2)$ and a worst-case space usage of $O(V)$.

(d) Letting $n = 500$ and $c = k/5$ for all integers $0 \leq k \leq 25$, use the function call `largestComponentSize( erdosRenyiGraph(n,c) )` to obtain the data necessary to make a nice figure showing how the size of the largest component varies with the mean degree. No credit if axes or lines on the figure are unlabeled.

You should see something surprising on your plot! What you should see is a *phase transition*, of the same kind that happens in physics, e.g., when water boils into vapor or when a ferric metal suddenly becomes magnetic. Below a *critical value* of c (which value does it look like on your plot?), the size of the largest component

is *independent* of the size of the graph, containing $\Theta(1)$ vertices, while above that value, they are proportional, so that the largest component contains $\Theta(n)$ vertices. The transition between these two behaviors occurs suddenly, as a function of the mean degree c. This behavior is why, in some complex systems, making things more interconnected can produce sudden changes in system behavior.



It seems that the *critical value* of our graph is 1. The largest component sizes of the graphs with average degrees less than 1 are $\Theta(1)$, while after 1, the largest component sizes are dependent on the average degree.

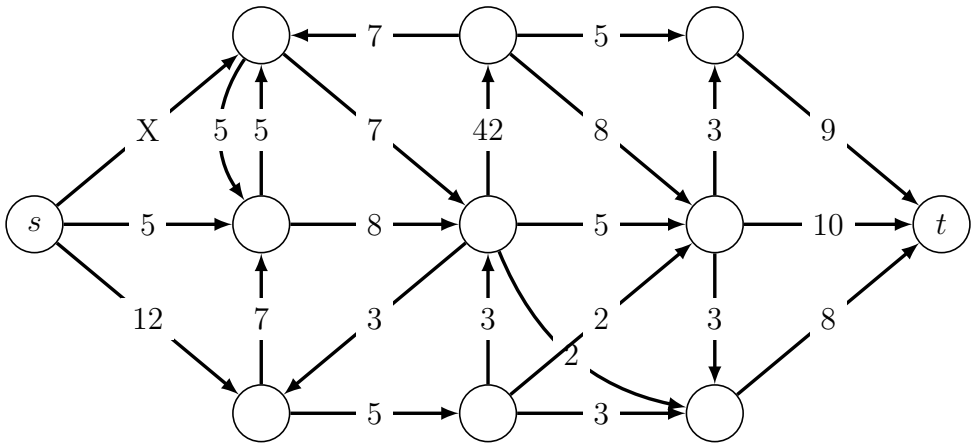
(e) (10 pts extra credit) Using the data file on the class Moodle and your functions above, compute the diameter and the mean geodesic distance. This file contains

a snapshot of the Facebook friendship network at a major U.S. university from 2005, when Facebook was only open to students, faculty, staff and alumni of about 100 select colleges and universities. Using the diameter and mean geodesic results for this graph, and your figure from part (1b), roughly estimate via extrapolation what the diameter and mean geodesic distance should be today, now that Facebook claims about $n = 10^9$ users. Discuss why the value of your rough estimate is or is not surprising to you in any way, and what it might mean for how information travels across Facebook.

Although we weren't able to obtain exact values for the diameter and the mean geodesic distance of the Facebook network due to time constraints and the script's lengthy runtime, here is how we approached the problem and how our implementation looked like:

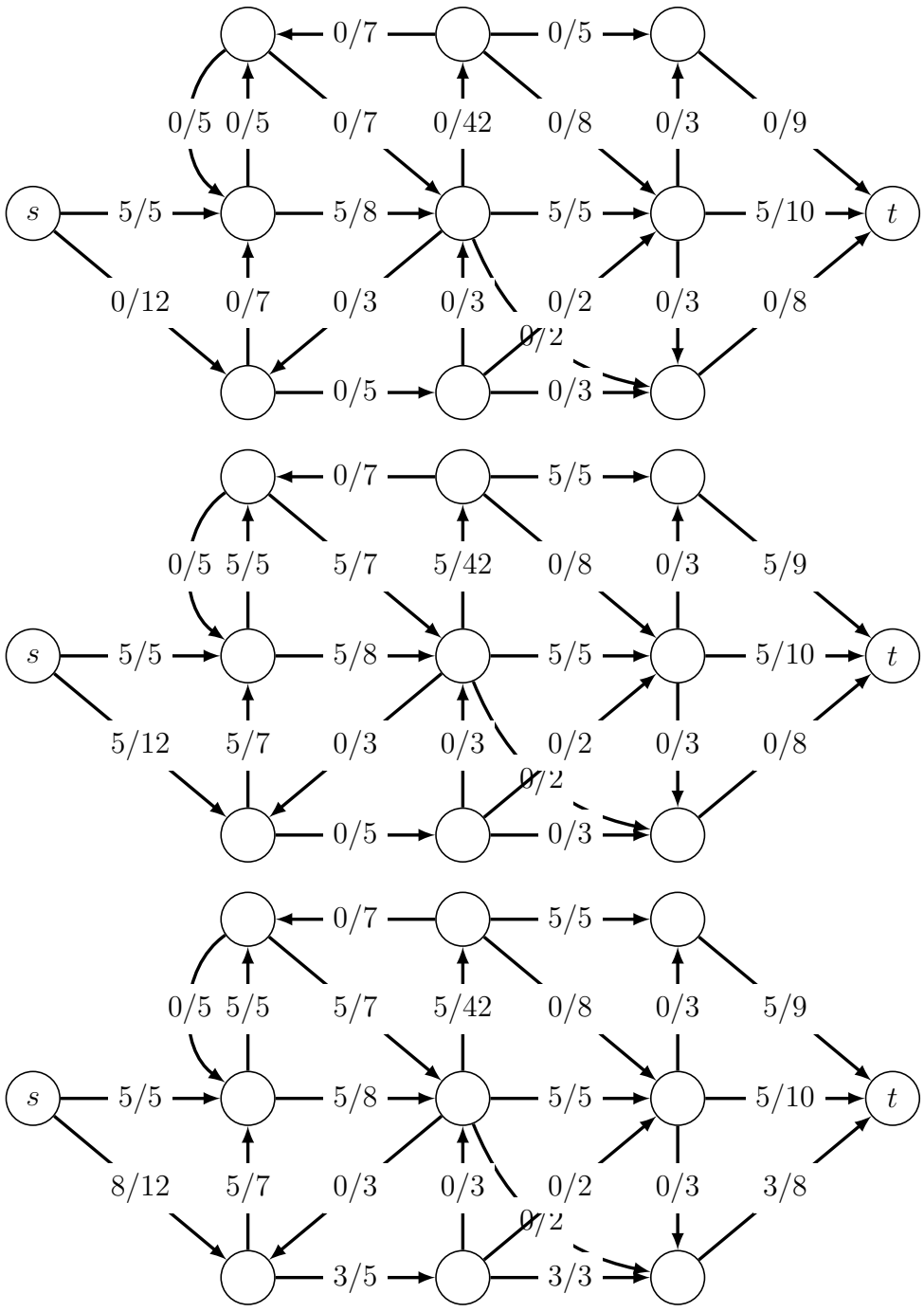
We tried to construct a graph using the data in the file, and after constructing the graph, we would pass the graph into our `howBigIsThisGraph` function to compute the diameter and mean geodesic distance. We used an array of arrays for the adjacency list representation of the graph, and as we read the data from the file, we added edges to our graph (both in the form (i, j) and (j, i)).

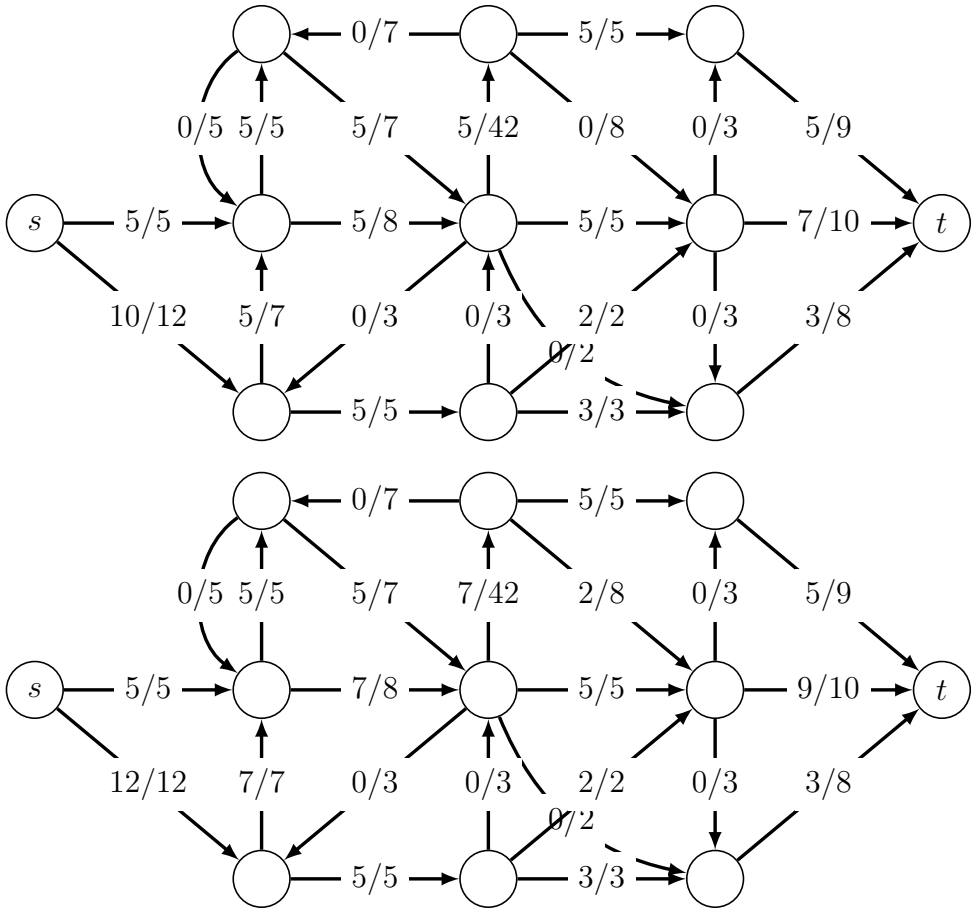
2. (30 pts total) Hagrid, the half-giant gamekeeper at Hogwarts, has installed a set small canals that convey water from a spring in the Forbidden Forest s to his house t. (He couldn't install just one big canal for some vague reason having to do with the Forbidden Forest getting mad about it.) Now, he's considering adding a new canal connecting the spring to his directed distribution network G. However, he's not sure how much additional water he will be able to push through G after adding the proposed canal. Hagrid needs your help to figure it out. The diagram below shows G', the network G plus the proposed canal X; edge labels indicate edge capacities.



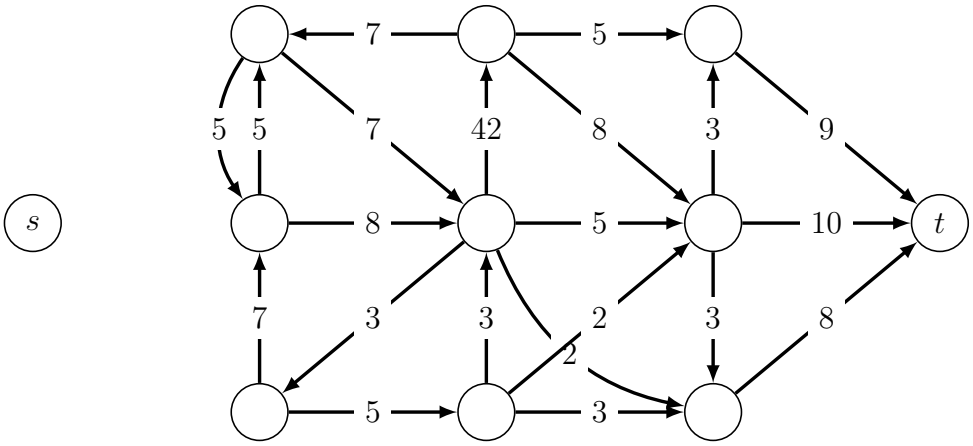
(a) Make a diagram showing the minimum cut corresponding to the maximum flow for G (where $X = 0$). Explain in terms of saturated and avoided edges why this is the minimum cut. Give the weight of this cut.

Using Ford-Fulkerson we get the following steps. The numbers listed are the residual capacities.


0/7
0/5
0/5
0/5
0/7
0/42
0/8
0/3
0/9
s
5/5
5/8
5/5
5/10
t
0/12
0/7
0/3
0/3
0/2
0/3
0/8
0/2
0/5
0/3
0/7
5/5
0/5
5/5
5/7
5/42
0/8
0/3
5/9
s
5/5
5/8
5/5
5/10
t
5/12
5/7
0/3
0/3
0/2
0/3
0/8
0/2
0/5
0/3
0/7
5/5
0/5
5/5
5/7
5/42
0/8
0/3
5/9
s
5/5
5/8
5/5
5/10
t
8/12
5/7
0/3
0/3
0/2
0/3
3/8
0/2
3/5
3/3

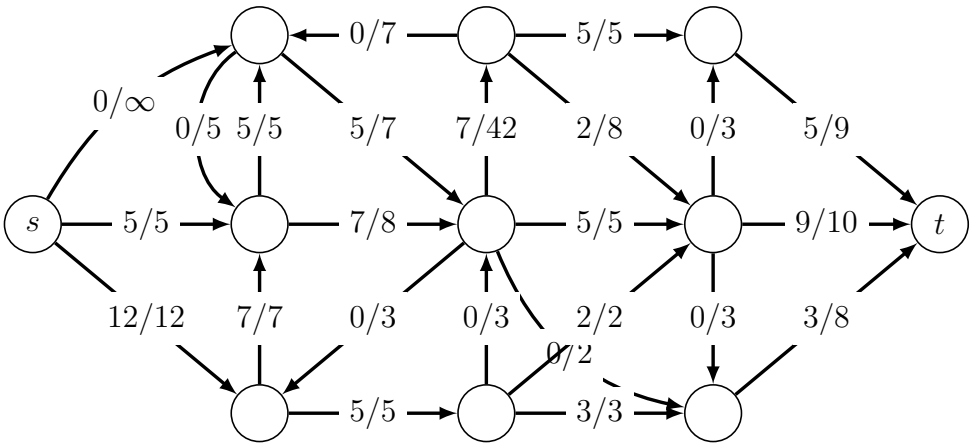


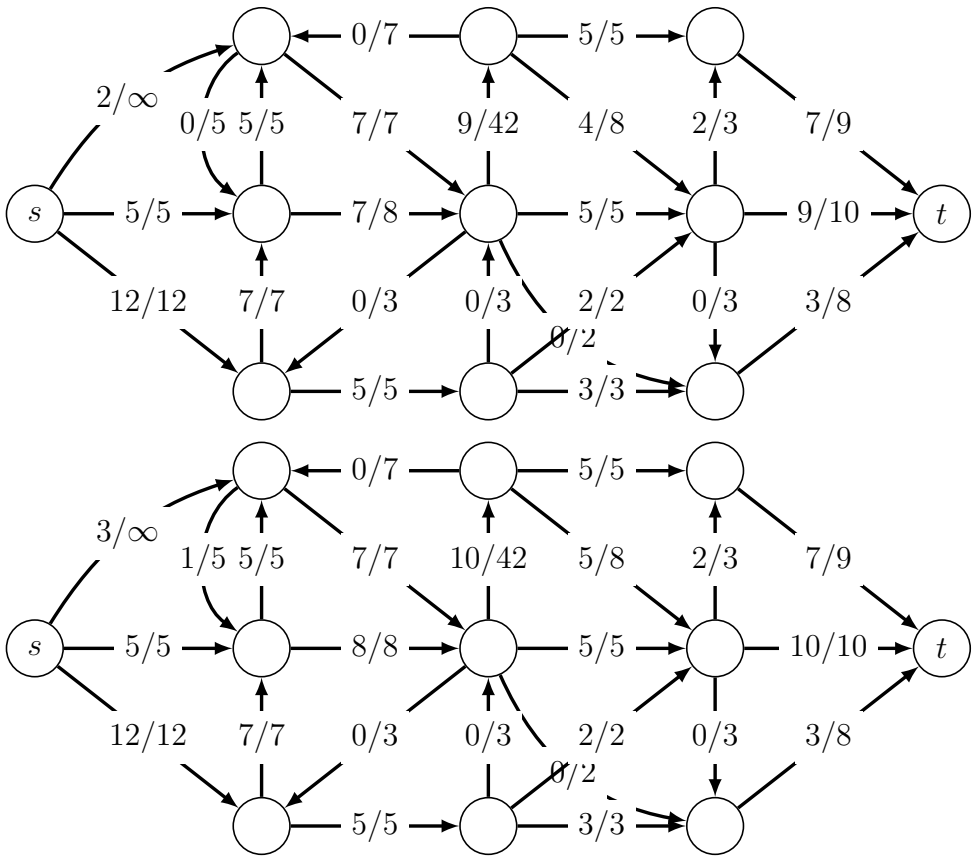
Note here that the min cut is right at the start point from the *start* node to the weight 5 edge and the weight 12 edge, thus placing s as the only node on the left side of the cut and every other node in the right side of the cut. These edges are saturated, and if you remove them it separates the entire graph into two sub-graphs. This cannot be said of any other set of edges in the graph, and it also happens to be a very ligthweight cut with weight $c(S,T) = 5 + 12 = \mathbf{17}$. Here is the diagram showing the cut:



(b) If Hagrid adds the canal X, what is the smallest capacity that would maximize the increase in the water flow across the network? Explain.

If we continue the algorithm, but include X with capacity ∞, then we will run the algorithm, and see the flow through the edge. This will give us the minimum capacity that will maximize increase in water flow.





Thus, the minimum capacity for X to maximize flow through the rest of the graph is **3**. The min cut is the three saturated nodes on top of each other with depth 2 and capacity 7, 8, and 5, leaving a total min cut of $7 + 8 + 5 = 3 + 5 + 12 = 20$.

(c) Describe how Hagrid could use a min-cut/max-flow algorithm to decide what capacity X should be used for an arbitrary graph $G = (V, E)$ and arbitrary proposed edge $(u, v) \notin E$ with capacity X.

In much same way we found the value of X in the graph above, the first thing to do would be to solve for the max flow of the graph without the edge included (or with edge weight 0). Then add in the edge, with infinite capacity, and proceed solving the algorithm. Then the flow running through the edge in question will be the maximum help that it can add to the graph. Note how this gives the same solution as solving the whole graph over again because you can add the paths in any order in the Ford-Fulkerson method, which means that it is the same as solving the graph as though the edge had infinite capacity the whole time, and we simply chose to not add any of its paths in until last.

3. (30 pts total) After a brilliant prank goes awry, your wizard friends Fred and George Weasley have bitter argument. You intervene to keep the peace and they agree to stay away from each other, for the time being. In particular, they have agreed that when navigating the halls of Hogwarts, each will not walk on any section of stone that the other wizard has stepped on that day. The wizards have no problem with their paths crossing at an intersection. The problem, however, is that they both still need to get to the Great Hall each day to eat. Fortunately, both the Gryffindor house entrance s and Great Hall t are at intersections. You have a map of Hogwarts' hallways and their intersections, on which the Gryffindor entrance and the Great Hall are marked.

   Your task is to determine whether there exists a pair of edge disjoint paths that would allow your friends to both get from s to t. Explain how to represent the problem as a graph G for which a straight-forward application of a max-flow/min-cut algorithm will yield the answer, and the paths.

   Note that if there only exists one edge out of Gryffindor or one edge into the Great Hall then there is no solution. Fortunately, we know that's not true because the problem states that they are at intersections, not dead-ends. However, if at any point in the graph we can remove an edge and make the graph disconnected, then we also know that they have to take that path to get to the Great Hall, and there's no way for them to both get there.

   This is what can be described as a bottleneck. In essence, multiple things are trying to get through something that isn't large enough to fit both of them. In this case, there are two things, namely Fred and George's paths. The thing that they're trying to get through is the path to Great Hall.

   Speaking a little more mathematically, let's say Fred's path and George's path each have a "width" of 1. Thus, in order for them to be side-by-side (they both go to the Great Hall in the same day), there must be a capacity of $1 + 1 = 2$.

   More specifically, if we represent Hogwarts as a graph, every edge has capacity 1, and Fred's and George's paths each have a flow of 1, since there is no overlap between edges of the two paths. When one edge is used, the other cannot use a fraction of it.

   Thus, if we run the min cut/max flow algorithm on this graph from Gryffindor to Great Hall, or s to t, respectively, then we need a max flow of at least 2 in order for them to both get to Great Hall without crossing paths. In this case, once we obtain a max flow of two, since it is a decision problem, we can halt the algorithm and return a "yes".

   More generally, if we finish the algorithm and calculate the max flow, the max flow, which is the min cut, is the number of people who can get to the Great Hall from

Gryffindor house without two people ever treading on the same stone. The min cut represents the nodes that, if removed, disconnect Gryffindor house from the Great Hall and make it so nobody can go through.

4. (10 pts extra credit) Preparing for a big end-of-semester party at Hogwarts, you crack open the Gryffindor cellar and count n bottles of fine drink. Dumbledore has previously warned you that exactly k of these bottles have been poisoned (he wouldn't go into detail as to how exactly this came to be), and consuming poisoned drink will result in an unpleasant death. The party starts in one hour, and you do not want to poison any of your guests.

   Luckily, a family of ℓ docile rats occupies a corner of the cellar, and they have graciously volunteered to be test subjects for identifying the poisoned bottles. Let $\ell = o(n)$ and $k = 1$, and assume it takes just under one hour for poisoned drink to kill a rat. (Hence, you only get one shot at solving this problem.)

   Describe a scheme by which you can feed the drink to rats and identify with complete certainty the poisoned bottle, prove that the scheme is correct, and give a tight bound on the number of rats ℓ necessary to solve the problem.

   Dumbledore's hint: 1010101111

   The most obvious solution would be to have n rats, each of which drinks from each drink to test it. Then the rat that dies, we know that the drink they had was poisoned. However, that requires n rats, which is a lot. The most obvious thing to do is to have no rat drink from the last barrel, and then if no rat dies, then we know the poisoned drink is the last barrel.

   For this solution what we are going to do is
   1) Don't assign a rat to the far barrel
   2) Divide the current partitions of the rest of the barrels into two partitions
   3) One rat drinks from the first half of the partitions
   4) Another drinks from the second half of the partitions
   5) Repeat step one with all current partitions

   Now this is far from technical code, but it has the same asymptotic number of rats as `binarySearch`'s runtime, only we assign two rats instead of making one comparison. Therefore the asymptotic number of rats needed is $\Theta(2 \lg n) = \Theta(\lg n) = o(n)$.

The following code is written in Python.

```
def computeDistances(graph, source):

    distances = [float("inf")] * len(graph)
    visited = [False] * len(graph)
    queue = []
    max_qlen = 0
    ops = 0

    distances[source] = 0
    queue.append(source)

    while queue:  #basic bfs

          u = queue.pop(0)
          ops += 1

          for v in graph[u]:
              if visited[v] == False:
                 visited[v] = True
                 ops += 1
                 distances[v] = distances[u] + 1
                 ops += 1
                 queue.append(v)
                 ops += 1

              if len(queue) > max_qlen:
                 max_qlen += 1

          visited[u] = True
          ops += 1

   return distances, (max_qlen, ops)
```

```
def largestComponentSize(graph):

    visited = [False] * len(graph)
    queue = []
    components = [0] * len(graph)  #will store component numbers
    component = 0
    max_component = 0

    def bfs(u):  #basic bfs method
        queue.append(u)
        while queue:
              u = queue.pop(0)

              for v in graph[u]:
                  if visited[v] == False:
                     visited[v] = True
                     components[v] = component
                     queue.append(v)

              visited[u] = True
              components[u] = component

    for u in range(len(graph)):
        if visited[u] == False:  #has to be in different component
           bfs(u)
           component += 1

    num_comp = [0] * (component)   #will store size of components

    for u in components:
        num = num_comp[u] + 1
        num_comp[u] = num
        if num > max_component:
           max_component = num

    return max_component
```

```
def howBigIsThisGraph(graph):

    diameter = 0
    sum_paths = 0   #sum of lengths of all paths
    num_paths = 0   #number of total paths
    ops = 0
    max_qlength = 0

    for u in range(len(graph)):

        distances = computeDistances(graph, u)[0]  #array stores distances from u
        ops += computeDistances(graph, u)[1][1]
        qlength = computeDistances(graph, u)[1][0]
        if qlength > max_qlength:
           max_qlength = qlength

        for i in distances:
            if i != float("inf") and i != 0:
               sum_paths += i
               num_paths += 1
               ops += 2

               if i > diameter:
                  diameter = i
                  ops += 1

    return (diameter, float(sum_paths)/num_paths), (ops, qlength)


def ErdosRenyiGraph(n, c):

    graph = [[] for i in range(n)] #allocate array of arrays (adjacency list)

    for u in range(n-1):
        for v in range(u+1, n):
            edge = random.randint(1, n-1 )
            if edge <= c :    #probability of c/(n-1)
               graph[u].append(v)    #add to graph edge (u,v)
               graph[v].append(u)    #add (v,u)
    return graph
```