CS 5854: Networks, Crowds, and Markets
Homework 2

Instructor: Rafael Pass TAs: Cody Freitag, Drishti Wali

Assigned: October 2, 2019 Due: October 27, 2019, 11:59 pm

Kushal Singh

Collaborators: I worked with Chris Shei and Scooter Blume.
Outside resources: I used the following outside resources: https://medium.com/100-days-of-algorithms/day-49-ford-fulkerson-e70045dafd8b.
Late days: I have used 1 many late days on this assignment.

Part 1: Best-Response Dynamics

1. (a) Let G_1 and G_2 be games with the same set of players and action sets for each player, and *assume that BRD converges to a PNE* in both of these games. Consider a game $G = (G_1 + G_2)$ where each player plays a single action for both G_1 and G_2 (i.e. plays the same action in both games at the same time) and receives utility equal to the *sum* of the utilities they would earn from G_1 and G_2. *Will BRD converge in this game?* If so, prove it; if not, find a counterexample and argue why it doesn't converge. (*Hint:* Start with a game that may not converge and try to split it into two separate games.)

   (b) Assume in a particular game $G = G_1 + G_2$ that BRD does converge. Will it necessarily converge to a state that is also a PNE in either G_1 or G_2? If so, prove it; if not, find a counterexample.

Solution:

(a) BRD will not necessarily converge in this game. Let's consider the Matching Pennies game, which we know does not converge to PNE. Specifically, let's model G as the Matching Pennies payoff matrix.

	$(*, H)$	$(*, T)$
$(H, *)$	(1, 0)	(0, 1)
$(T, *)$	(0, 1)	(1, 0)

Let's now try to decompose G into two separate games, G_1 and G_2. Let G_1 be the game with the following payoff matrix:

	$(*, H)$
$(H, *)$	(1, 0)
$(T, *)$	(0, 1)

Let G_2 be the game with the following payoff matrix:

	$(*, T)$
$(H, *)$	(0, 1)
$(T, *)$	(1, 0)

From above, we can see that G_1 converges at (H, H), while G_2 converges at (T, T). Since $G = G_1 + G_2$, which does not converge, we have found our counterexample.

(b) No, it may not necessarily converge to a state that is also a PNE in either G_1 or G_2.

	$(*, H)$	$(*, T)$
$(H, *)$	(15, 15)	(8, 8)
$(T, *)$	(0, 0)	(-15, -15)

From above, we can see that G_1 converges at (H, H).

	$(*, H)$	$(*, T)$
$(H, *)$	(-15, -15)	(8, 8)
$(T, *)$	(0, 0)	(15, 15)

From above, we can see that G_2 converges at (T, T).

	$(*, H)$	$(*, T)$
$(H, *)$	(0, 0)	(16, 16)
$(T, *)$	(0, 0)	(0, 0)

But, game $G = G_1 + G_2$ converges at (H, T) which is not a PNE in either G_1 or G_2.

□

2. Consider the process of "better-response dynamics" rather than BRD, where instead of choosing a player's best response (i.e. the response that maximizes their utility given others' actions), a player chooses any response that *strictly* increases their utility given other players' actions.

   (a) Does the process of "better-response dynamics" still converge in games for which there is an ordinal potential function Φ? Prove it or show a counterexample. (*Hint:* This is in the notes now. It suffices to reference the correct theorem.)

   (b) Can you think of a game for which BRD converges but "better-response dynamics" might not? Show an example or justify that one doesn't exist. (*Hint:* Remember that the better-response dynamics graph can have edges that the BRD graph might not. What if those edges formed a loop?)

* **Bonus Question 1.** Recall the Traveler's Dilemma that we studied in chapter 1. We have already illustrated why BRD converges in this game; find a **weakly** ordinal potential function Φ over states of this game and use it to definitively prove the convergence of BRD. Make sure to justify that the potential function you find is weakly ordinal (you can do this via computer if you wish).

* **Bonus Question 1.5.** (*This may be very difficult!*) Determine whether *better-response dynamics* converge for the Traveler's Dilemma. No formal proof is necessary, and you need not come up with a potential function. You are free to either use simulations, show (experimentally) that the graph contains no cycle, or find an ordinal potential function and either prove it or experimentally verify.

Solution:

(a) Yes, "better-response dynamics" will converge if and only if there is an ordinal potential function. The proof for this is shown in Theorem 3.6.

(b) Below is an example of a game where BRD converges, but "better-response dynamics" does not.

	$(*, A)$	$(*, B)$	$(*, C)$
$(a, *)$	(2, 1)	(1, 2)	(3, 3)
$(b, *)$	(1, 2)	(2, 1)	(3, 3)

In this example, best response dynamics converges at (a, C), and (b, C). But, if we start at (a, A) and follow better response dynamics, we may not reach a convergence. Consider the following scenario, where Player 1 represents the player across the columns, while Player 2 represents the player along the rows.

At (a, A), Player 1 may switch to playing B, which brings us to (a, B).
At (a, B), Player 2 may switch to playing b, which would bring us to (b, B).
At (b, B), Player 1 may switch to playing A, which would bring us to (b, A).
At (b, B), Player 2 may switch to playing a, which would bring us back to (a, A), thus restarting the entire sequence.

□

Part 2: Networked Coordination Games

3. Consider the following simple coordination game between two players:

	$(*, X)$	$(*, Y)$
$(X, *)$	(x, x)	(0, 0)
$(Y, *)$	(0, 0)	(y, y)

Show how we can pick x and y and then modify this payoff matrix by adding an intrinsic utility for a single player and a single choice (e.g. give player 1 some intrinsic utility for choice Y) such that the socially optimal state of the game is no longer an equilibrium.

Solution: Let's start with x = -1, and y = 1.

	$(*, X)$	$(*, Y)$
$(X, *)$	(-1, -1)	(0, 0)
$(Y, *)$	(0, 0)	(1, 1)

Now, let's suppose we give Player 1 an intrinsic utility of +10 for choice X. Then, we get the following payoff matrix:

	$(*, X)$	$(*, Y)$
$(X, *)$	(9, -1)	(0, 0)
$(Y, *)$	(0, 0)	(1, 1)

From the above payoff matrix, we can see that the action profile (X, X) is the socially optimal state of the game, since its total welfare is -1 + 10 = +9, whereas (X, Y) and (Y, X) each have total welfare of 0 + 0 = 0, and (Y, Y) has total welfare of 1 + 1 = 2. However, at (X, X), Player 2's best response strategy would be to switch to choice Y, since 0 > -1. This leads us to the action profile (X, Y). But, this encourages Player 1 to switch to Y as well, since 1 > 0, bringing us to the PNE (Y, Y). The action profile (Y, Y), however, only has a social welfare of 1 + 1 = +2, which is less than the socially optimal outcome of +9 seen at (X, X). Therefore, the socially optimal outcome of the game is no longer an equilibrium. □

4. Consider a networked coordination game on a complete graph of five nodes. Assume for simplicity that all edges represent the same coordination game (that is, $Q_e(X,X) = Q_{e'}(X,X)$ for any pair of edges e, e', and respectively for Y, but it is not necessarily the case that $Q_e(X,X) = Q_e(Y,Y)$), and that all nodes have the same intrinsic values for X and Y (that is, $R_i(X) = R_j(X)$ for any nodes i, j, and respectively for Y).

   (a) Show a possible assignment of intrinsic and coordination utilities such that every pure-strategy Nash equilibrium of the resulting game must be socially optimal. Justify that this holds for your construction.

   (b) Is there a possible assignment of intrinsic and coordination utilities such that there exists a pure-strategy Nash equilibrium with social welfare less than half of the maximum attainable social welfare? If not, prove it. If so, show such an assignment and explain why your construction does not violate Theorem 4.5 from the notes (the price of stability). (*Hint:* Theorem 4.5 only talks about the *best* equilibrium. Try to make a simple two-node game with one very good equilibrium and one bad equilibrium, and then extend what you find there to the five-node graph. It also won't be necessary to use intrinsic values for this part.)

Solution:

   (a) Let's consider the following possible assignment of intrinsic utilities and coordination utilities. We set the intrinsic utility of X as 10, the intrinsic utility for Y as 0, and the coordination utilities of $(X,X) = 0$, $(Y,Y) = 0$, and $(X,Y) = 0$.

   Since the joint coordination utilities are all equal to 0, we can see that the intrinsic utility that is gained by each player is all that matters. Furthermore, the action profile that represents the best response for each player (also, the only PNE), is when all players choose X (the socially optimal PNE). The social welfare is the summation of the intrinsic utility (playing (X) or (Y)) and the coordination utility ((X,X), (X,Y), (Y,Y)). From above, since we know that the coordination utilities are zero, the only contribution toward the social welfare is the intrinsic value of X.

   (b) Yes, there is a possible assignment of intrinsic and coordination utilities such that there exists a pure-strategy Nash equilibrium with social welfare less than half of the maximum attainable social welfare.

   Let's consider the following coordination utilities: $(Y,Y) = 10$, $(X,X) = 1$, $(X,Y) = 0$.

   From these utilities, we can see that when all players choose X (PNE(X)) or all players choose Y (PNE(Y)), then this results in a PNE. When everyone plays Y, we've also reached the socially optimal outcome. The social welfare of PNE(X)$= 1 = \frac{1}{10}$th the social welfare of PNE(Y) $= 10$. We have shown an instance where there exists a PNE with social welfare less than half the maximum attainable social welfare, namely $\frac{1}{10}$th that of the maximum attainable social welfare.

This does not violate Theorem 4.5 from the notes because the price of stability only bounds the gap between social welfare and the **best** equilibrium. The gap between social welfare and the **worst** equilibrium would be the price of anarchy.

□

Part 3: Cascading Behavior in Networks

5. Consider the network in Figure **??**. Suppose that each node starts with the behavior Y, and has a $q = 2/5$ threshold for adopting behavior X. (That is, if at least $2/5$ of a node's neighbors have adopted X, that node will as well.)

   (a) Let e and f form a set S of initial adopters of X. Which nodes will eventually switch to X? (Assume that S will not participate in BRD.)

   (b) Find a cluster of density $1 - q = 3/5$ in the part of the graph outside S which blocks X from spreading to all nodes starting from S.

   (c) Add one node to S such that a complete cascade will occur at the threshold $q = 2/5$. Demonstrate how the complete cascade could occur (i.e. in what order nodes will switch).

   Solution:

   (a) Since k is connected to both e and f, which form the set S of initial adopters, we can see that $2/4 = 1/2$ of k's neighbors have adopted X. Since $q = 2/5 < 1/2$, k will be the first node to switch to behavior X. As a result of k's switch, node l now has $2/3$ of its neighbors playing X. Since $q = 2/5 < 2/3$, l will also switch to behavior X.

   Aside from the four nodes in the inner cluster (e, f, k, l), none of the other nodes on the periphery (c, d, i, n, m, h) will switch to X because they each have 2 other neighbors playing Y, and only one neighbor playing X. This means that each of the periphery nodes have a $q' = 1/3$ threshold for adopting behavior X. Since $q' < q$, these nodes will not switch.

   Nodes g and j won't adopt behavior X either since each of them only has one neighbor playing strategy Y. This means that g and j have a $q'' = 0$ threshold for adopting behavior X. Since $q'' < q$, these nodes will not switch.

   (b) The periphery nodes comprising c, h, m, n, i, d blocks X from spreading to all nodes starting from S. These periphery nodes have a density $= 2/3$. Since the requirement for the blocking set is that the density be $\geq (1 - q) = 3/5$, we can see that the $2/3$ density of the blocking set satisfies this criterion and, thus, prevents X from spreading to all nodes starting from S.

   (c) Adding either h or i will cause a complete cascade to occur at the threshold $q = 2/5$. If we add h, then...
   1) g will switch since its only neighbor is h.
   2) k will switch, as shown in part (a).
   3) l will switch, also shown in part (a). c will switch, since its neighbors (h, e) are now playing X, making its threshold $q' = 2/3 > q$ to switch to X. m will also switch, since its neighbors (h, k) are now playing X, making its threshold $q'' = 2/3 > q$ to switch to X.
   4) The same reasoning from step 3) can be applied for nodes d and n, causing d and n to switch to playing X as well.
   5) Then, i will switch, since both its neighbors (d, n) are playing strategy X.
   6) Finally, j will switch, since its only neighbor (i) is playing strategy X.

□

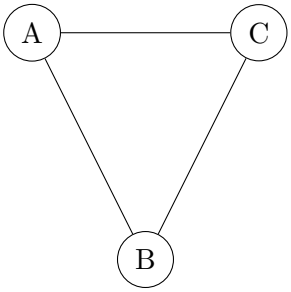
6. (a) Formulate a graph G, threshold q, and set S of initial adopters such that, assuming we start with S choosing X (and, importantly, able to participate in BRD) and other nodes choosing Y, we can either end up with every node in G playing X or with every node in G playing Y, depending on the order of switches in the BRD process.

   (b) What must be true of the set density of S for the above property to hold?

   (c) What must be true of the set density in any subset of $V \setminus S$ for the above property to hold?

Solution:

(a) Let's consider the graph G below. Let the set $S = \{A\}$, which means that A is playing the strategy X. Let nodes A and C play Y.



Let $q = 1/3$. If we run Best Response Dynamics at node A, then since node A has more than $1/3$ of its neighbors playing X, it will switch to playing X as well. Since C has more than $1/3$ of neighbors playing X, it also switches to playing X.

If we start BRD at node B, then since all of B's neighbors are playing Y, it will switch to playing Y because the threshold to switch to playing $Y = 2/3$ is greater than the threshold to remain playing $X = 1/3$.

(b) In order for the above property to hold, the set S must have a density less than q. This would mean that S is not strongly connected, thereby allowing a cascade by Y if BRD starts at a node in S.

(c) For the above property to hold, the set density in any subset of $V \setminus S$ should be less than $1 - q$, so that it can be cascaded by S.

□

Part 4: Traffic Networks

7. There are two cities, A and B, joined by two routes which pass through towns C and D respectively. There are 120 travelers who begin in city A and must travel to city B, and may take the following roads:

   - a local street from A to C with travel time $15 + x$, where x is the number of travelers using it,
   - a highway from C to B with travel time 90,
   - a highway from A to D with travel time 90, and
   - a local street from D to B with travel time $15 + y$, where y is the number of travelers using it.

   (a) Draw the network described above and label the edges with their respective travel times. The network should be a directed graph (assume that all roads are one-way). Note: you may just describe the graph if you are typing the assignment up.

   (b) Find the Nash equilibrium values of x and y. (Show that this is the equilibrium.)

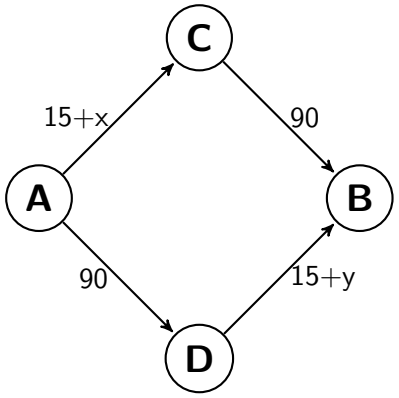
   (c) The government now adds a new *two-way* road connecting the nodes where local streets and highways meet. This new road is extremely efficient and requires no travel time. Find the new Nash equilibrium.

   (d) What happens to the total travel time as a result of the availability of the new road? (You don't need to explain, a calculation is fine.)

   (e) Now suppose that the government, instead of closing the new road, decides to assign routes to travelers to shorten the total travel time. Find the assignment that minimizes the total travel time, and determine the total travel time using this assignment. (*Hint:* It is possible to achieve a total travel time less than the original equilibrium. Remember that, with the new road, there are now four possible routes that each traveler can take.)

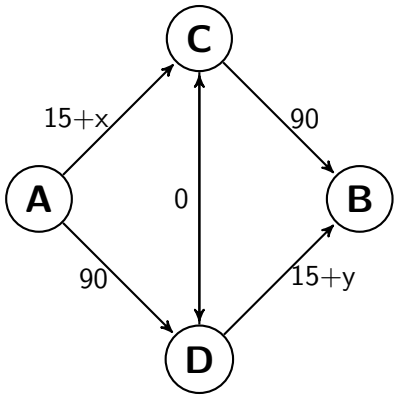
Solution:

(a) Network Graph



(b) The Nash equilibrium values would be $x = y = 60$. This will give each traveler a total travel time = 165, since $ACB = 75 + 90 = 165$, and $ADB = 90 + 75 = 165$. We can show that this is an equilibrium because if a traveler who originally follows the route ADB decides to switch and instead follow the route ACB, then the total travel time for ACB would increase to 166, while the total travel time for ADB would decrease to 164. Then, a traveler who originally follows the route ACB will observe that following the route ADB is now the best response strategy, thus leading to the equilibrium values $x = y = 60$.

(c) Network Graph



The Nash equilibrium values would be $x = y = 75$. With the added route between C and D, travelers would go from $A \implies C$ and then $D \implies B$, until the number of travelers causes the routes to slow down. After 75 travelers take this route, then all routes take a travel time of 180 and the preferable route becomes $A \implies D \implies C \implies B$. Applying the same reasoning as part a), we can see that the equilibrium values occur when $x = y = 75$, because for any traveler taking either the above or below route, switching would result in either the same or worse travel time.

(d) Without the new road, the total travel time $= 120 \times 65 = 19,800$.
With the new road, the total travel time $= 120 \times 180 = 21,600$.
From the calculations above, we can see that the total travel time increases with the availability of the new road.

(e) The assignment that minimizes the total travel time is when $x = y = 38$. We can find this by essentially splitting the problem in half. First, we calculate the minimum travel time leaving A and going to either C or D. We can write this as the following equation:

$$T(x) = (15 + x)x + 90(120 - x)$$
$$T(x) = x^2 - 75x + 10800 = 0$$
$$x = 37.5$$

By symmetry, we can see that going from $C/D \implies B$ is the same problem in reverse, so the minimum $y = 37.5$. Rounding up, we can calculate the total travel time as $(15 + 38) \times 2) \times 38 + 180 \times (120 - 38) = 18788$.

□

Part 5: Experimental Evaluations

8. This problem will make use of the data set available at:

   [http://snap.stanford.edu/data/egonets-Facebook.html].

   In particular, please refer to the file "facebook_combined.txt.gz"; it contains a text file listing all 88,234 edges (*undirected* edges representing Facebook friendship) in a sampled 4,039-node network (nodes are numbered 0 to 4038). It will be useful, for problem 8 to be able to input a graph presented in such a format into your code.

   (a) Consider the contagion examples that we observed in chapter 5 of the notes. Given an *undirected* graph, a set of early adopters S, and a threshold q (such that a certain choice X will spread to a node if more than q fraction of its neighbors are playing it), produce an algorithm that permanently infects the set S of early adopters with X and then runs BRD on the remaining nodes to determine whether, and to what extent, the choice will cascade through the network. (*Note:* "BRD" in this case is simply the process of iteratively deciding whether there is a node that will switch its choice and performing this switch.)

   Once again, turn in your code and verification that your algorithm works on a few simple test cases. In particular, include the output on the two examples from Figure 4.1 in the notes; let S be the set of nodes choosing X in the figure, and, for each of the two graphs, include one example with a complete cascade and one without one (and specify what value of q you used for each).

   (b) Run your algorithm several (100) times on a fairly small random set of early adopters ($k = 10$) with a low threshold ($q = 0.1$) on the Facebook data set. What happens? Is there a complete cascade? If not, how many nodes end up being "infected" on average?

   (c) Run your algorithm several (10) times with different values of q (try increments of 0.05 from 0 to 0.5), and with different values of k (try increments of 10 from 0 to 250). Observe and record the rates of "infection" under various conditions. What conditions on k and q are likely to produce a complete cascade in this particular graph, given your observations?

   * **Bonus Question 2.** (Optional, extra credit awarded depending on quality of solution.) Design an algorithm that, given a graph and a threshold q, finds (an approximation of) the smallest possible set of early adopters that will cause a complete cascade. Try running it on the Facebook data with different values of q and seeing how large a set we need.

Solution:

(a) **Output for Figure 3.1a:**
For $q = 0.5$, there is a full cascade where 1 node switches.
For $q = 1$, we do not have a full cascade. This one was an interesting scenario because there is only one node that is not an early adopter, while both its neighbors are.

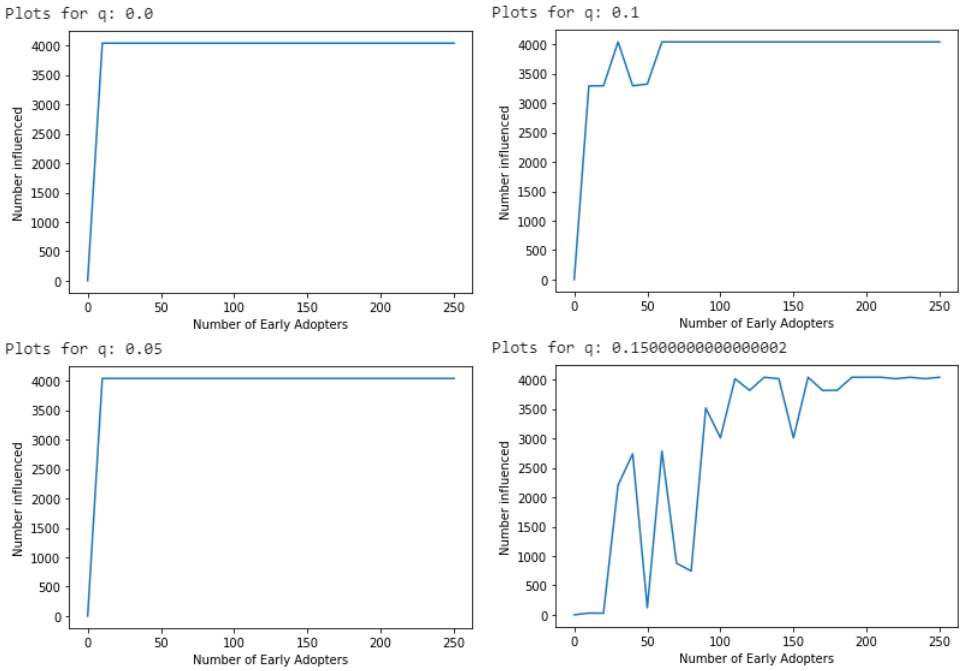
Output for Figure 3.1b:
For $q = 0.3$, we see that 4 nodes end up switching, resulting in a full cascade.
For $q = 0.34$, no nodes end up switching, so we do not have a full cascade.

(b) Running the algorithm 100 times on the Facebook data with a threshold of $q = 0.1$ leads to an average node change of 3239.73. From the results in the output below, we can see that there are several complete cascades (namely, the ones where 4039 nodes have changed), but there are also instances when significantly less nodes change (i.e. 17, 15, 43).

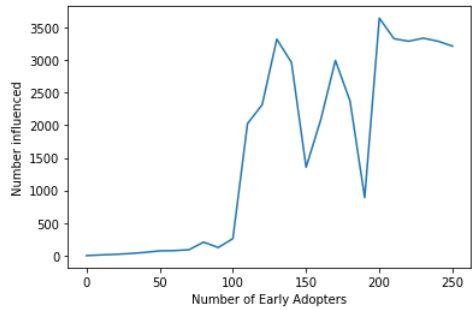
```
[16] print ("average with q=.1 after 100 iterations is: ",sum(num_influenced)/len(num_influenced))
     for i, val in enumerate(num_influenced):
       print(val, end=' ')
       if ( not i % 20):
         print('')
```

```
average with q=.1 after 100 iterations is:  3239.73
4039
4039 3291 3291 3292 3291 3293 72 3290 3290 3292 3296 3290 3292 3292 3290 4039 3291 3294 3294 4039
3292 3292 3293 3290 3291 3293 20 3291 3291 3291 4039 4039 3291 3290 39 3296 3291 3291 3292 15
4039 3293 3292 3292 3321 3291 3293 4039 3293 3294 3319 3292 3292 3291 3291 3290 3290 3293 4039 4039
3290 3290 3290 4039 3292 3291 3291 17 3292 4039 4039 3293 43 3290 3290 3320 3293 4039 3291 3291
3290 3293 3295 3290 3292 3291 3295 4039 4039 3292 4039 3293 3321 3291 4039 3320 3294 3294 4039
```

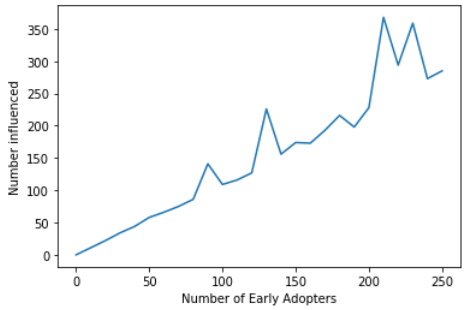
(c) From the plots below, we can see that when $q <= 0.1$ and $k >= 60$, then a full cascade is likely to result in the graph. When q is lower, then k can also be lower because this means that if a particular node is connected to an early adopter, then its likelihood of switching is higher. The most likely combination of a complete cascade is when q is low and k is high, because this means that there are a lot of early adopters and they are very likely to influence their neighbors, since q is low.



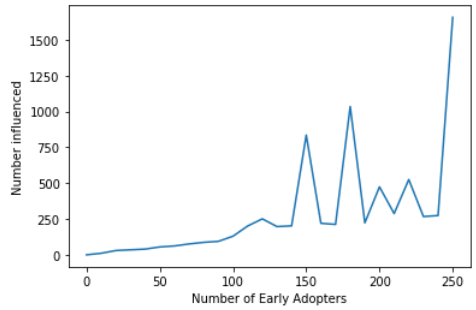
Plots for q: 0.2



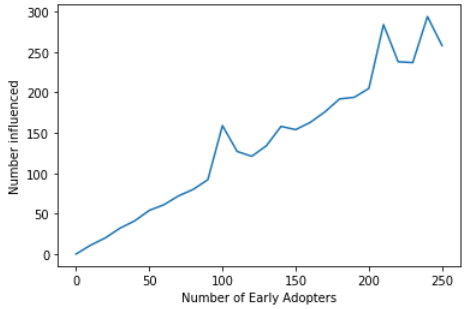
Plots for q: 0.30000000000000004



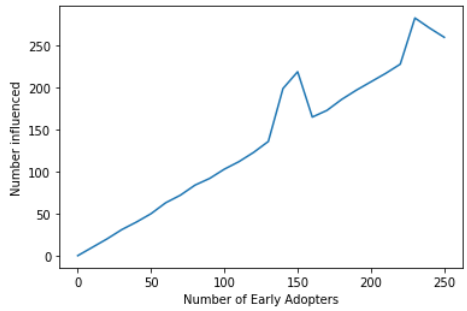
Plots for q: 0.25



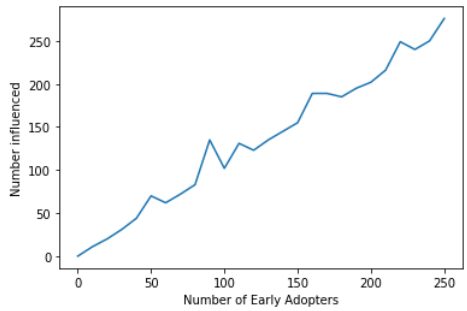
Plots for q: 0.35000000000000003



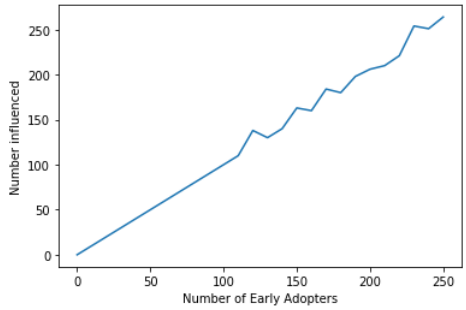
Plots for q: 0.4



Plots for q: 0.45



Plots for q: 0.5



□

9. Consider the following problem: There are n Uber drivers and m potential riders. At a fixed point in time, each driver has a list of compatible riders that she can pick up. Our goal will be to match drivers to riders such that the most riders at this time are picked up. We will use the maximum-flow algorithm, described in Chapter 6 of the notes to do this.

   (a) First, implement an algorithm that, given a *directed* graph, a source s, a sink t, and edge capacities over each edge in E, computes the maximum flow from s to t (you must implement this algorithm yourself). Turn in your code and verify that your algorithm works on a few simple test cases. In particular, test your algorithm on the graphs in Figures 6.1 and 6.3 from the lecture notes and submit the output.

   (b) Given a set of n drivers, m riders, and sets of possible riders that each driver can pick up,

      i. explain how we can use this maximum-flow algorithm to determine the the maximum *number* of matches, and

      ii. explain how we can additionally extend this to actually find the matchings.

      (*Hint:* See the notes if you are confused about how to do this.)

   (c) Implement a maximal matching algorithm for Uber drivers and riders. Specifically, given n drivers with constraints specified on m riders, computed the assignments of drivers to riders. Test your algorithm on at least 2 examples (with at least 5 riders and drivers each). Explain your examples and your results.

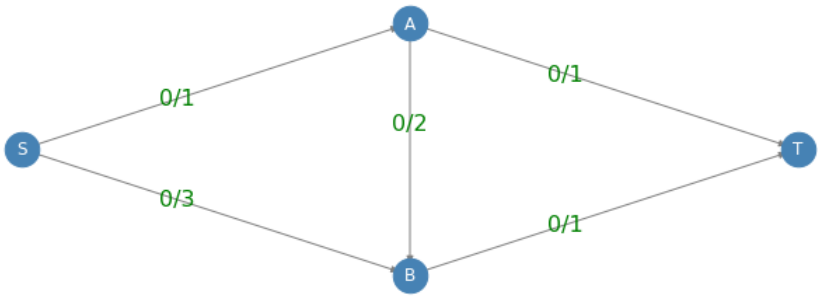
   (d) Now consider the case where there are n drivers and n riders, and the drivers each driver is connected to each rider with probability p. Fix $n = 1000$ (or maybe 100 if that's too much), and estimate the probability that all n riders will get matched for varying values of p. Plot your results.

   * **Bonus Question 3:** In the above example, let $p^*(n)$ be the smallest value of p where all n riders get matched with at least 99% probability. Prove bounds on what $p^*(n)$ is as a function of n, e.g. is $p^*(n) \geq 1/n$ or $p^*(n) \leq 1/2$. You will get partial credit for providing experimental evidence towards a proposed idea.

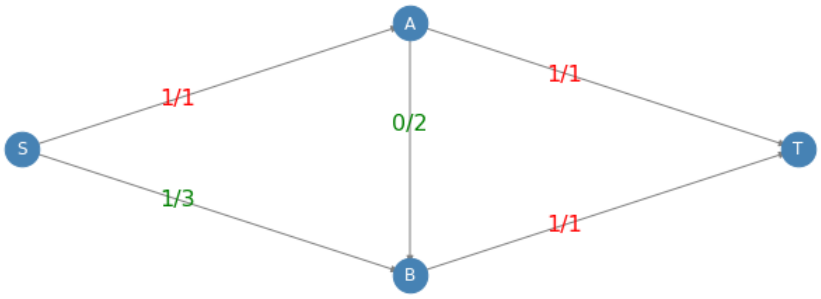
Solution:

(a) **Output for Figure 6.1:**

Original Graph



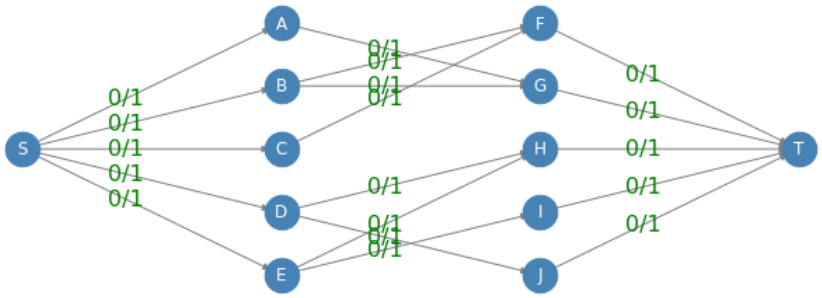
Max Flow Graph



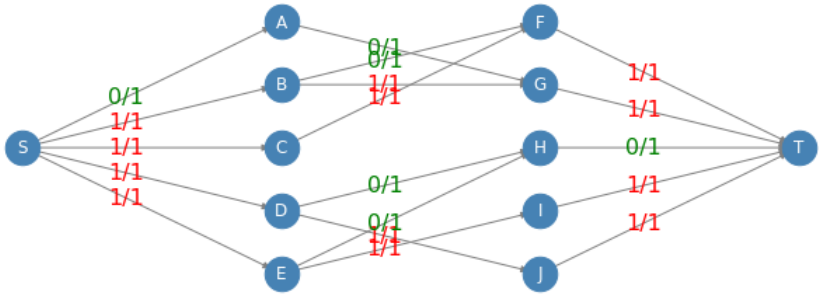
Max Flow 2

Output for Figure 6.3:

Original Graph

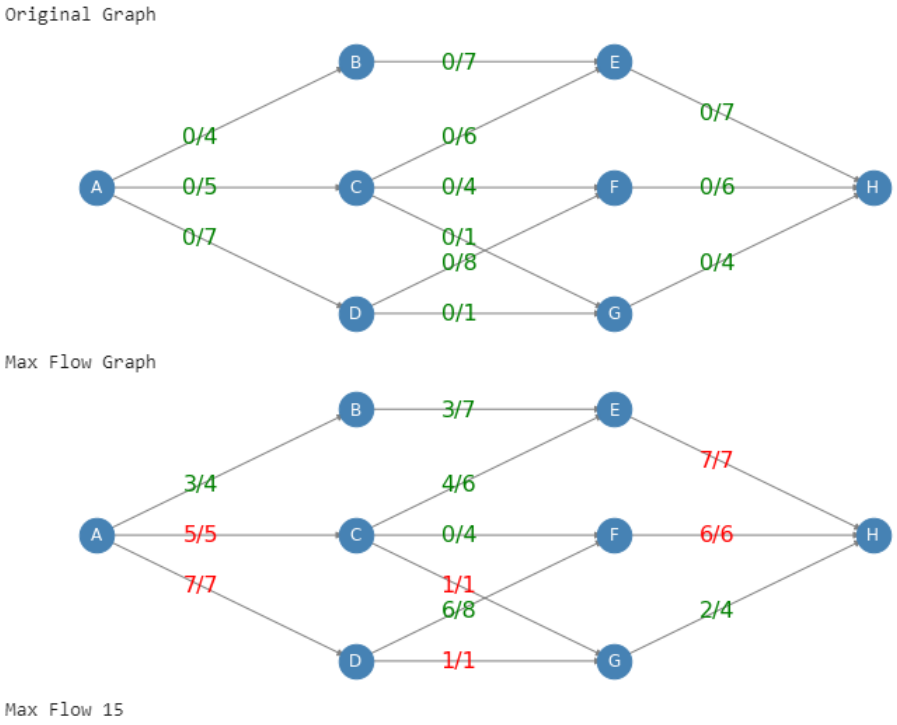


Max Flow Graph



Max Flow 4

Extra test:

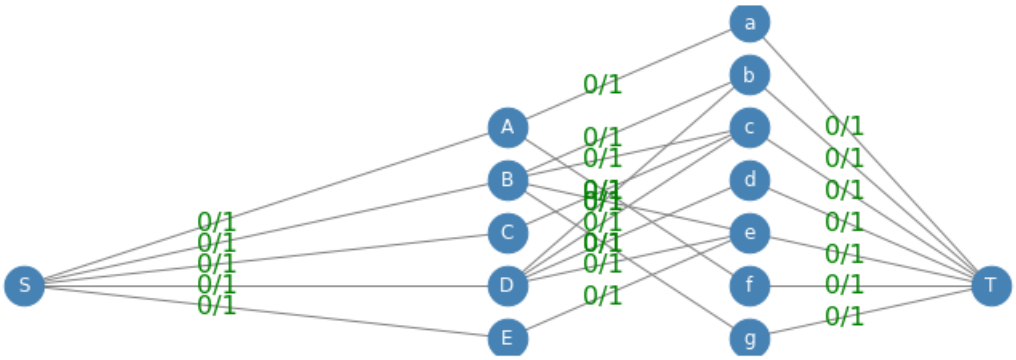


(b) i. We can use the maximum flow algorithm to determine the maximum number of matches by running max flow on a graph where there are edges between a source S and all the drivers with a weight of 1, edges between the drivers and riders with a weight of 1, and edges between the riders and a sink T with a weight of 1, and seeing how many edges between drivers and riders are saturated. The number of saturated edges is equivalent to the maximum number of matches.

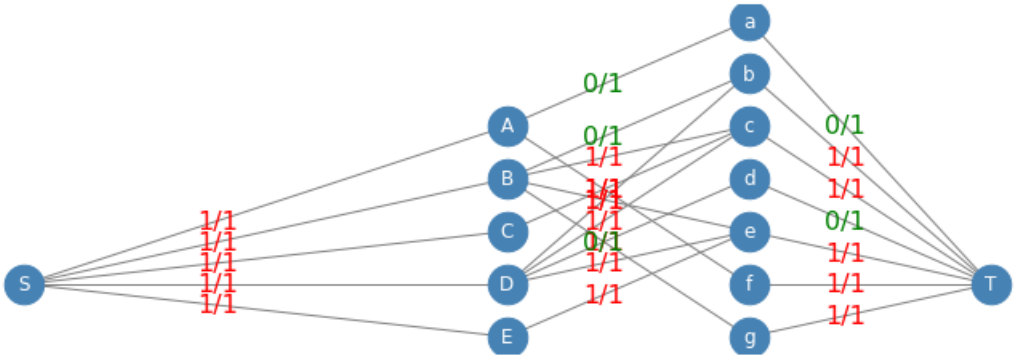
ii. After running the max flow algorithm, we can trace/follow the backward edges in the residual graph from T to S to actually find the matchings.

(c) **Example 1:** With $n = 5$ drivers and $m = 7$ riders, our max flow algorithm returned 5 since there are at most 5 possible matches. Two of the riders were unable to be paired because of the bottleneck in the number of drivers. Furthermore, not all drivers were able to match with all the riders. The way our max flow algorithm works is that by keeping track of the residual graph, it can assign matches based on individuals with the most constraints. In our example, E only matches with e, but e can potentially match with B, D as well. In order for us to send the maximum flow, we would want to send one unit of flow along the edge (E, e), since B, D have significantly more options of riders to be paired with. If E and e were not matched, then we wouldn't have a maximal match.

Original Graph

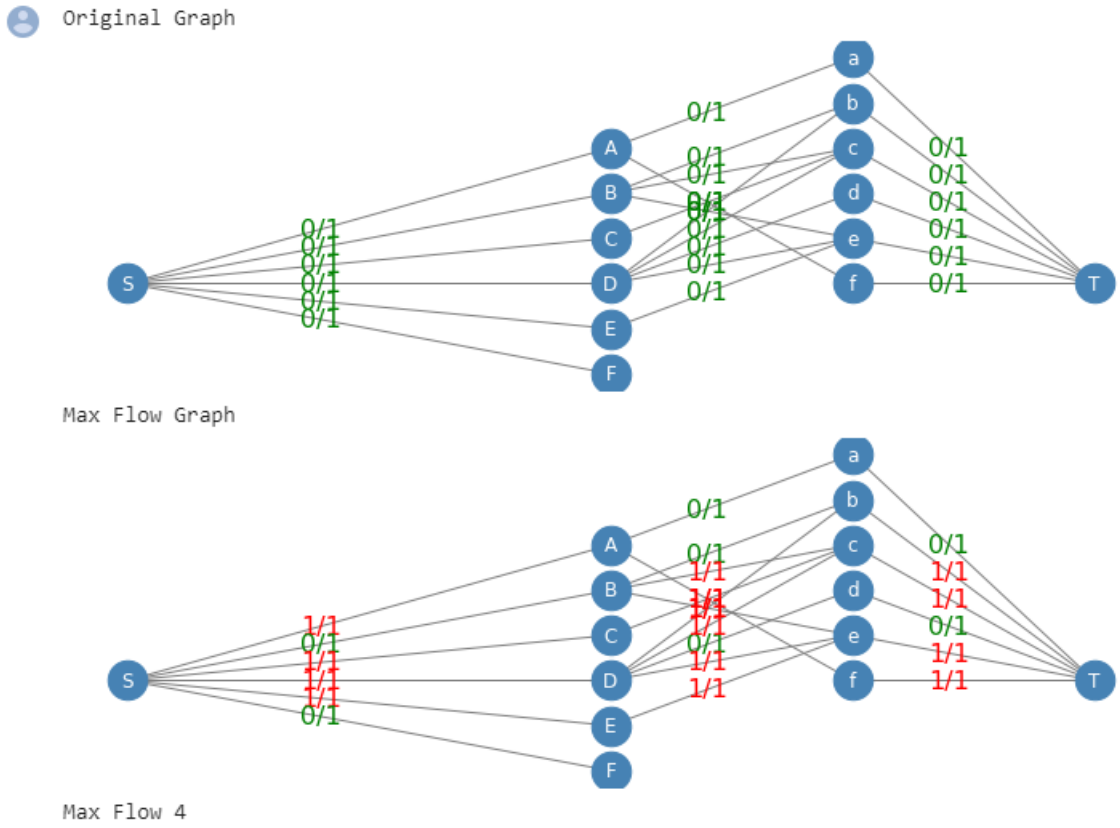


Max Flow Graph

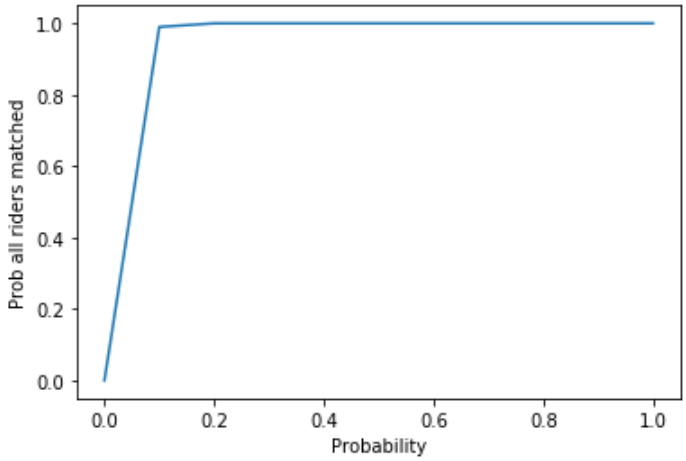


Max Flow 5

Example 2: In this case we have $n = 6$ drivers and $m = 6$ riders. Since all drivers are not connected to all the riders (e.g. F), we can't expect to get a maximal matching of 6. The end result is a maximal matching of 4.



(d) When $n = 100$, we can see that when $p >= 0.1$, the probability that all riders are matched shoots up to 100 percent. This graph is shown below.



```
import re
import random
import numpy as np
import networkx as nx
from networkx.algorithms import bipartite

import matplotlib.pyplot as plt
%matplotlib inline
```

8A

Consider the contagion examples that we observed in chapter 5 of the notes. Given an undirected gra adopters S, and a threshold q (such that a certain choice X will spread to a node if more than q fractic neighbors are playing it), produce an algorithm that permanently infects the set S of early adopters w runs BRD on the remaining nodes to determine whether, and to what extent, the choice will cascade t network. (Note: “BRD” in this case is simply the process of iteratively deciding whether there is a nod its choice and performing this switch.)

```
def contagion_brd(G, S, q):
  if (S != -1):
    G = create_graph(G, random.sample(range(0, 4038), S))
  num_changed = 0
  flag = True
  prev = 0
  while (flag):

    for item in G.items():
      early_adopter_bool = item[1][0]

      if (not early_adopter_bool):
        neighbors = item[1][1]
        if (len(neighbors) == 0):
          break

        count = 0
        for neighbor in neighbors:
          neighbor_bool = G[neighbor][0]

          if (neighbor_bool):
            count = count + 1

        fraction = count / len(neighbors)


        if (fraction > q):
          elem = item[0]

          if (G[elem][0] == False):
            num_changed = num_changed + 1
          G[elem][0] = True

    #check if we have converged yet and update the flag
    if (num_changed == prev):
      flag = False
    else:
      prev = num_changed
```

```
  return num_changed, G

# A -- B -- C -- D
test_case_1 = {
    'A': [True, ['B']],
    'B': [True, ['A', 'C']],
    'C': [False, ['B', 'D']],
    'D': [True, ['C']]
}


#     B    D    F
#     |    |    |
#A -- C -- E -- G
test_case_2 = {
    'A': [True, ['C']],
    'B': [True, ['C']],
    'C': [True, ['A', 'B', 'E']],
    'D': [False, ['E']],
    'E': [False, ['C', 'D', 'G']],
    'F': [False, ['G']],
    'G': [False, ['E', 'F']],
}

y = contagion_brd(test_case_2,-1, 0.33)
y
#z = test_contagion(test_case_1, 0.10)
#z
```

```
(4,
 {'A': [True, ['C']],
  'B': [True, ['C']],
  'C': [True, ['A', 'B', 'E']],
  'D': [True, ['E']],
  'E': [True, ['C', 'D', 'G']],
  'F': [True, ['G']],
  'G': [True, ['E', 'F']]})
```

8B

Run your algorithm several (100) times on a fairly small random set of early adopters (k = 10) with a l 0.1) on the Facebook data set. What happens? Is there a complete cascade? If not, how many nodes “infected” on average?

```
def create_graph(fb_graph, S):
  new_graph = dict()

  for edge in fb_graph:
    i = edge[0]
    j = edge[1]

    if (i not in new_graph.keys()):
      if (int(i) in S):
        new_graph[i] = [True, []]
      else:
        new_graph[i] = [False, []]

    if (j not in new_graph.keys()):
      if (int(j) in S):
```

```
        new_graph[j] = [True, []]
      else:
        new_graph[j] = [False, []]

    new_graph[i][1] = new_graph[i][1] + [j]
    new_graph[j][1] = new_graph[j][1] + [i]

  return new_graph


data_path = 'facebook_combined.txt'

with open(data_path) as f:
  raw_data = f.read()
  z = [s.strip().split(' ') for s in raw_data.splitlines()]

num_influenced = list()
for i in range(0, 100):
  influenced, res_graph = contagion_brd(z, 10, 0.1)
  num_influenced = num_influenced + [influenced + 10]


print ("average with q=.1 after 100 iterations is: ", sum(num_influenced)/len(num_influence
for i, val in enumerate(num_influenced):
  print(val, end=' ')
  if (not i % 20):
    print('')
```

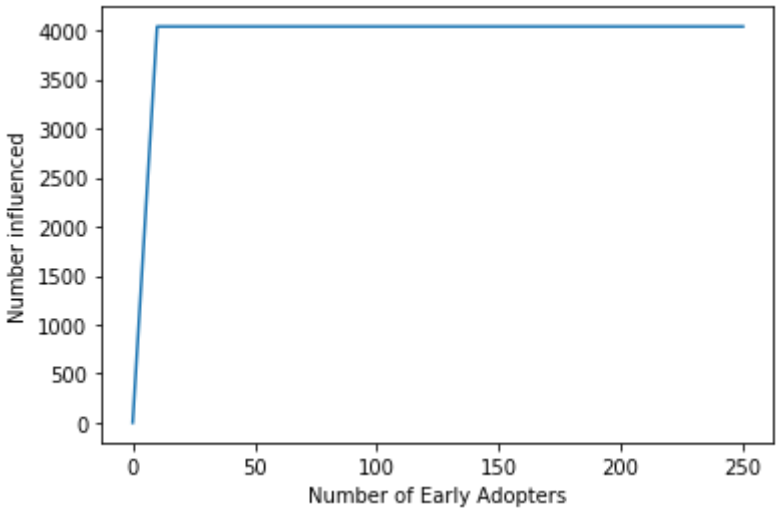
```
average with q=.1 after 100 iterations is:  3239.73
4039
4039 3291 3291 3292 3291 3293 72 3290 3290 3292 3296 3290 3292 3292 3290 4039 32
3292 3292 3293 3290 3291 3293 20 3291 3291 3291 4039 4039 3291 3290 39 3296 3291
4039 3293 3292 3292 3321 3291 3293 4039 3293 3294 3319 3292 3292 3291 3291 3290
3290 3290 3290 4039 3292 3291 3291 17 3292 4039 4039 3293 43 3290 3290 3320 3293
3290 3293 3295 3290 3292 3291 3295 4039 4039 3292 4039 3293 3321 3291 4039 3320
```

8C

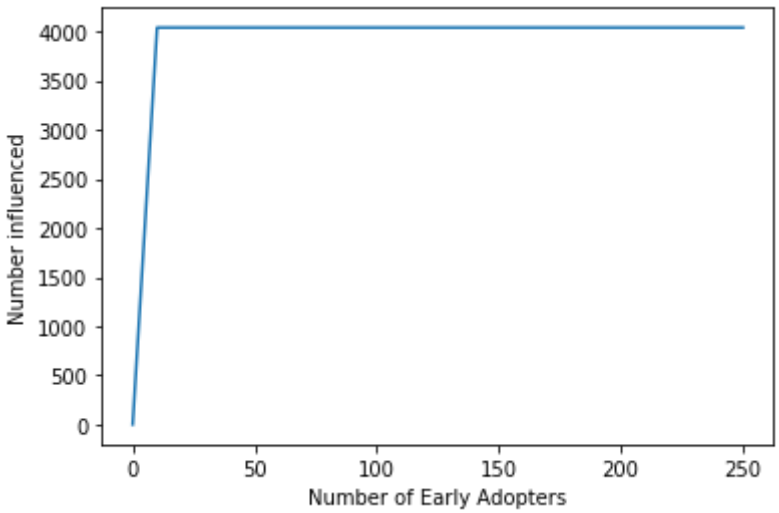
Run your algorithm several (10) times with different values of q (try increments of 0.05 from 0 to 0.5), different values of k (try increments of 10 from 0 to 250). Observe and record the rates of “infection” conditions. What conditions on k and q are likely to produce a complete cascade in this particular gra observations?

```
diff_k_list = list(np.arange(0, 260, 10))

for diff_q in np.arange(0, 0.55, 0.05):
  print("Plots for q: {}".format(diff_q))
  num_influenced_list = []

  for diff_k in np.arange(0, 260, 10):
    influenced = contagion_brd(z, diff_k, diff_q)[0]
    num_influenced_list = num_influenced_list + [diff_k + influenced]

  plt.plot(diff_k_list, num_influenced_list)
  plt.xlabel("Number of Early Adopters")
  plt.ylabel("Number influenced")
  plt.show()
```

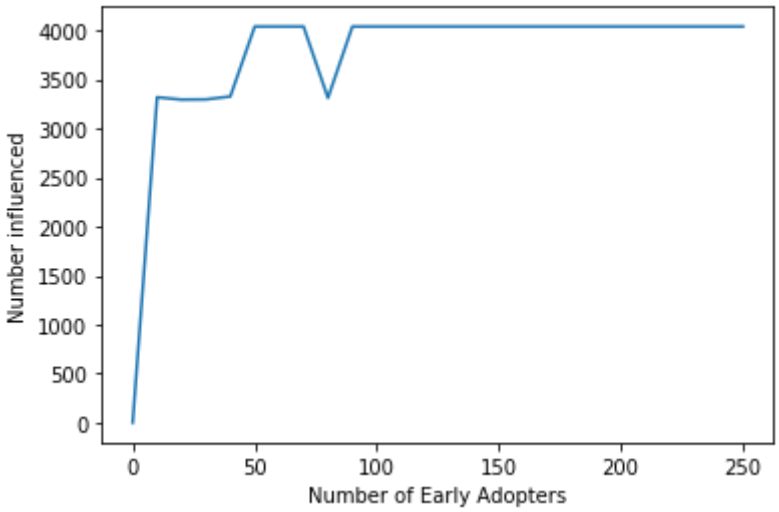
Plots for q: 0.0



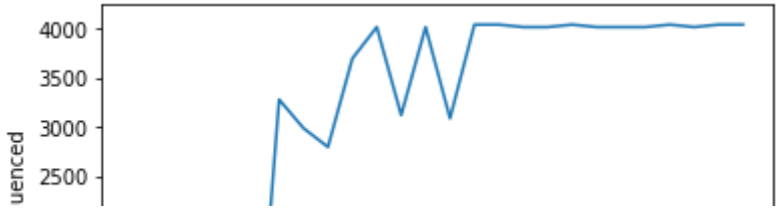
Plots for q: 0.05

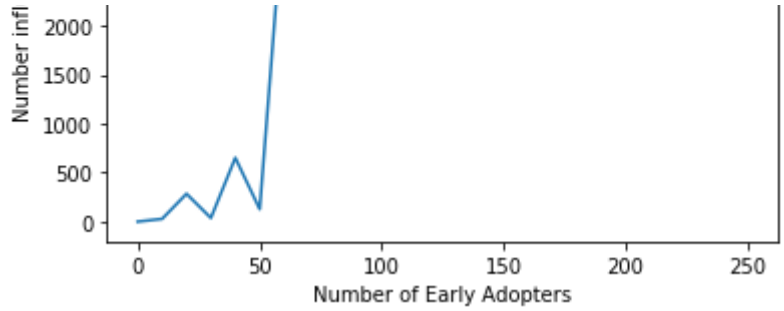


Plots for q: 0.1

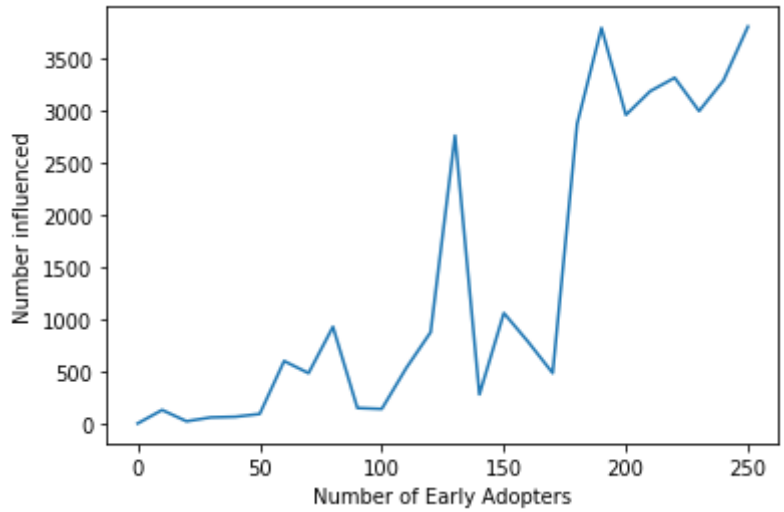


Plots for q: 0.15000000000000002

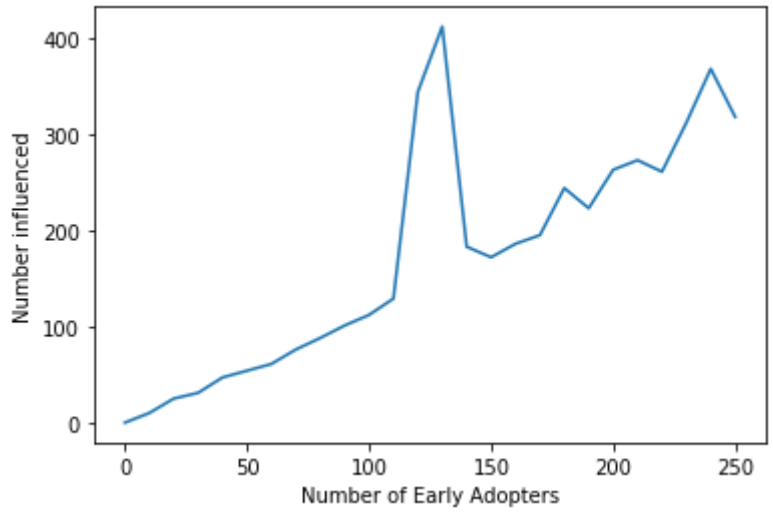




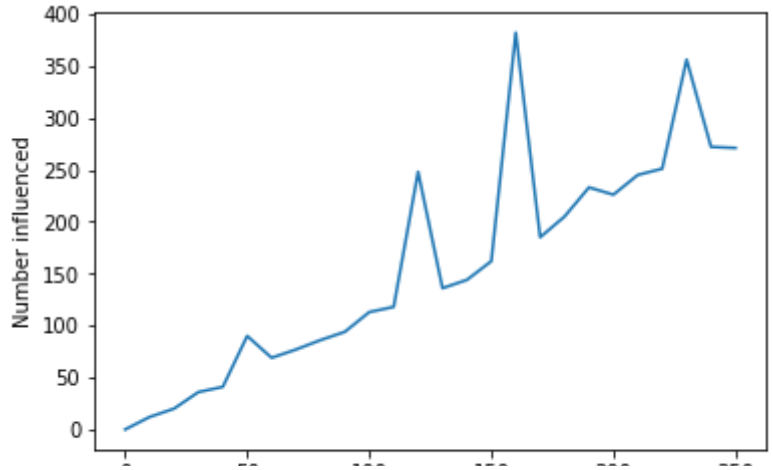

```
Plots for q: 0.2
```

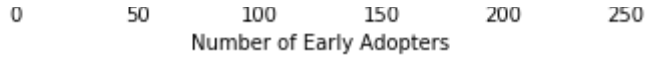



```
Plots for q: 0.25
```

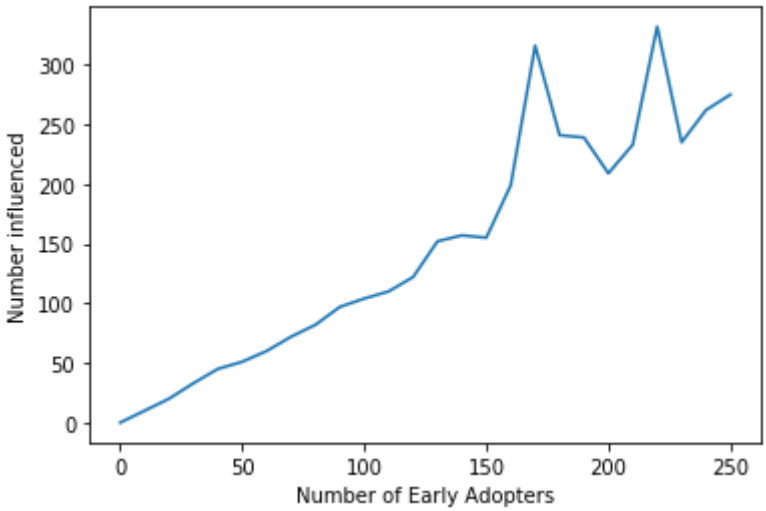



```
Plots for q: 0.30000000000000004
```

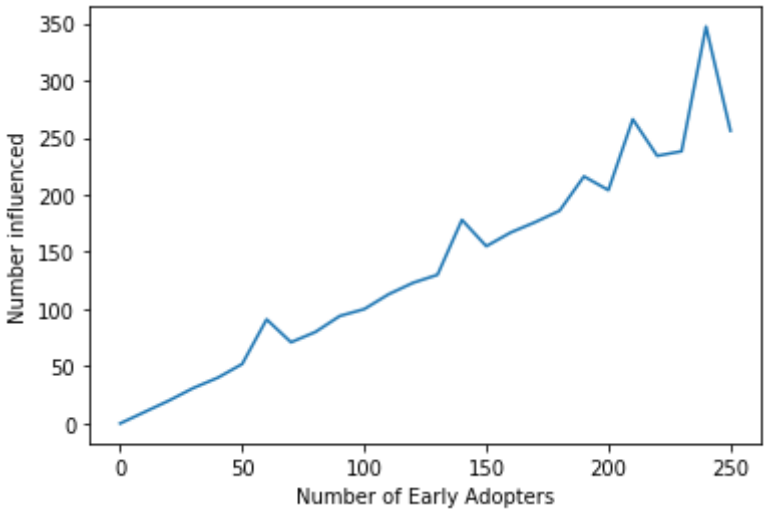




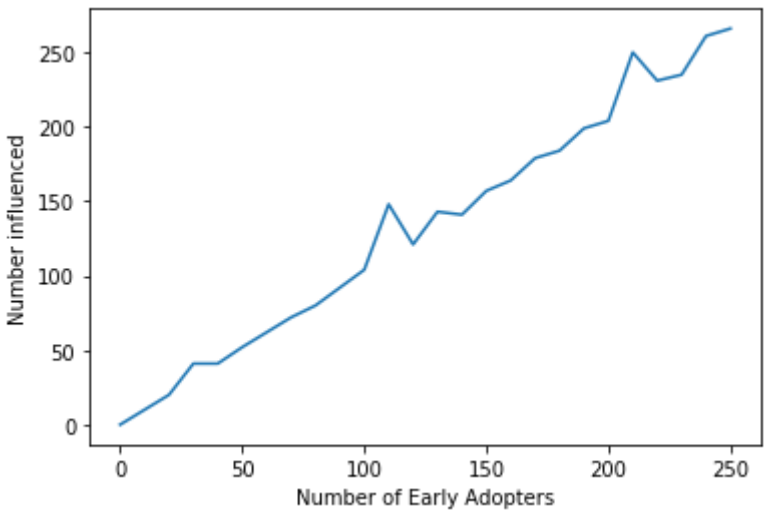

```
Plots for q: 0.35000000000000003
```



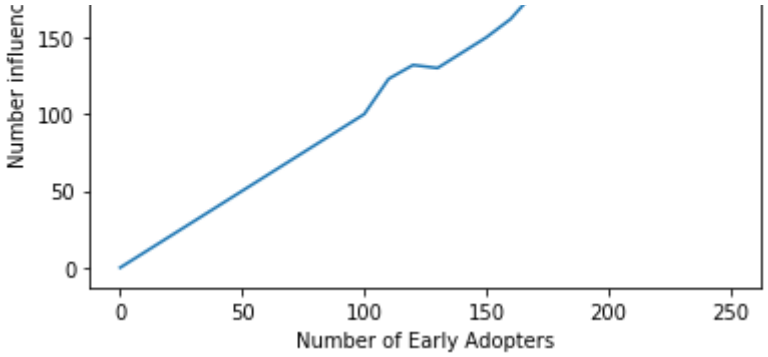

```
Plots for q: 0.4
```




```
Plots for q: 0.45
```




```
Plots for q: 0.5
```



Bonus Question 2. (Optional, extra credit awarded depending c of solution.)

Design an algorithm that, given a graph and a threshold q, finds (an approximation of) the smallest pc early adopters that will cause a complete cascade. Try running it on the Facebook data with different seeing how large a set we need.

9a

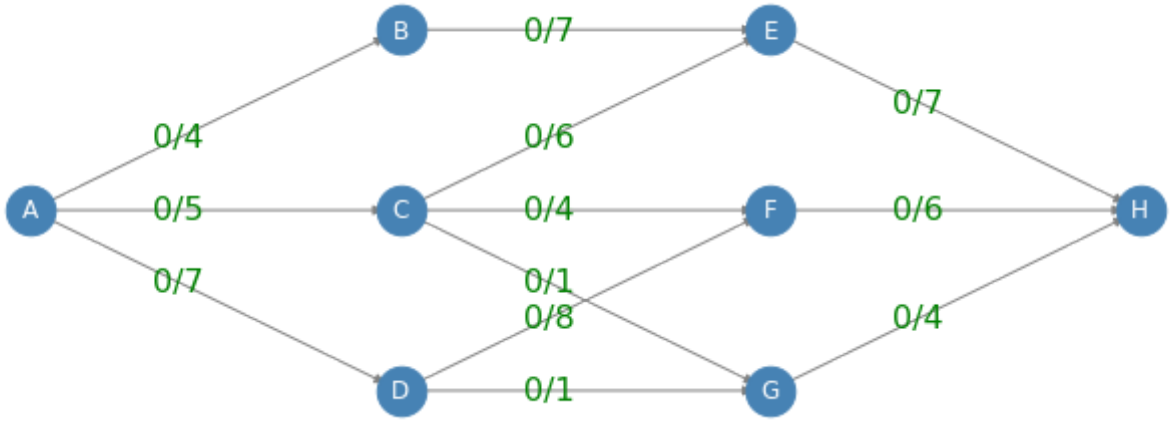
First, implement an algorithm that, given a directed graph, a source s, a sink t, and edge capacities ov E, computes the maximum flow from s to t (you must implement this algorithm yourself). Turn in you that your algorithm works on a few simple test cases. In particular, test your algorithm on the graphs and 6.3 from the lecture notes and submit the output.

```
#Implementation courtesy of the link below
#https://medium.com/100-days-of-algorithms/day-49-ford-fulkerson-e70045dafd8b
def ford_fulkerson(graph, source, sink, debug=None):
    flow, path = 0, True

    while path:
        path, reserve = depth_first_search(graph, source, sink)
        flow += reserve

        for v, u in zip(path, path[1:]):
            if graph.has_edge(v, u):
                graph[v][u]['flow'] += reserve
            else:
                graph[u][v]['flow'] -= reserve
    return flow


#def max_flow(G, s, t, c):
def depth_first_search(graph, source, sink):
    undirected = graph.to_undirected()
    explored = {source}
    stack = [(source, 0, dict(undirected[source]))]

    while stack:
        v, _, neighbours = stack[-1]
```

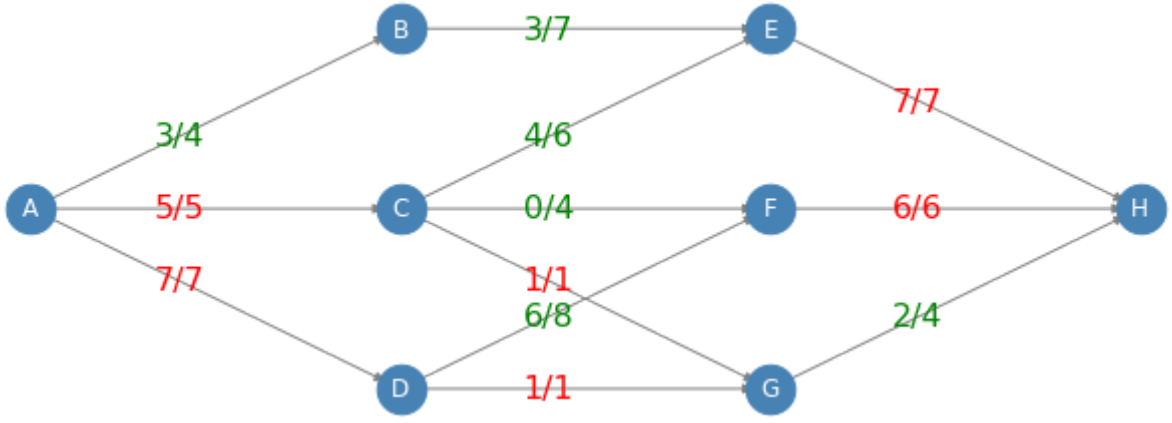
```
            if v == sink:
                break

            while neighbours:
                u, e = neighbours.popitem()
                if u not in explored:
                    break
            else:
                stack.pop()
                continue

            in_direction = graph.has_edge(v, u)
            capacity = e['capacity']
            flow = e['flow']
            neighbours = dict(undirected[u])

            if in_direction and flow < capacity:
                stack.append((u, capacity - flow, neighbours))
                explored.add(u)
            elif not in_direction and flow:
                stack.append((u, flow, neighbours))
                explored.add(u)

        reserve = min((f for _, f, _ in stack[1:]), default=0)
        path = [v for v, _, _ in stack]

        return path, reserve



def flow_debug(graph, path, reserve, flow):
    print('flow increased by', reserve, 'at path', path, '; current flow', flow)


def draw_graph(layout, graph):
    plt.figure(figsize=(12, 4))
    plt.axis('off')

    nx.draw_networkx_nodes(graph, layout, node_color='steelblue', node_size=600)
    nx.draw_networkx_edges(graph, layout, edge_color='gray')
    nx.draw_networkx_labels(graph, layout, font_color='white')

    for u, v, e in graph.edges(data=True):
        label = '{}/{}'.format(e['flow'], e['capacity'])
        color = 'green' if e['flow'] < e['capacity'] else 'red'
        x = layout[u][0] * .6 + layout[v][0] * .4
        y = layout[u][1] * .6 + layout[v][1] * .4
        t = plt.text(x, y, label, size=16, color=color,
                     horizontalalignment='center', verticalalignment='center')

    plt.show()


graph = nx.DiGraph()
graph.add_nodes_from('ABCDEFGH')
graph.add_edges_from([
    ('A', 'B', {'capacity': 4, 'flow': 0}),
    ('A', 'C', {'capacity': 5, 'flow': 0}),
    ('A', 'D', {'capacity': 7, 'flow': 0}),
    ('B', 'E', {'capacity': 7, 'flow': 0}),
    ('C', 'E', {'capacity': 6, 'flow': 0}),
    ('C', 'F', {'capacity': 4, 'flow': 0}),
    ('C', 'G', {'capacity': 1, 'flow': 0}),
    ('D', 'F', {'capacity': 8, 'flow': 0}),
    ('D', 'G', {'capacity': 1, 'flow': 0}),
    ('E', 'H', {'capacity': 7, 'flow': 0}),
    ('F', 'H', {'capacity': 6, 'flow': 0}),
    ('G', 'H', {'capacity': 4, 'flow': 0}),
```

```
])

layout = {
    'A': [0, 1], 'B': [1, 2], 'C': [1, 1], 'D': [1, 0],
    'E': [2, 2], 'F': [2, 1], 'G': [2, 0], 'H': [3, 1],
}


print("Original Graph")
draw_graph(layout, graph)

max_test_graph = ford_fulkerson(graph, 'A', 'H', flow_debug)
print("Max Flow Graph")
draw_graph(layout, graph)
print("Max Flow", max_test_graph)
```

```
Original Graph
```



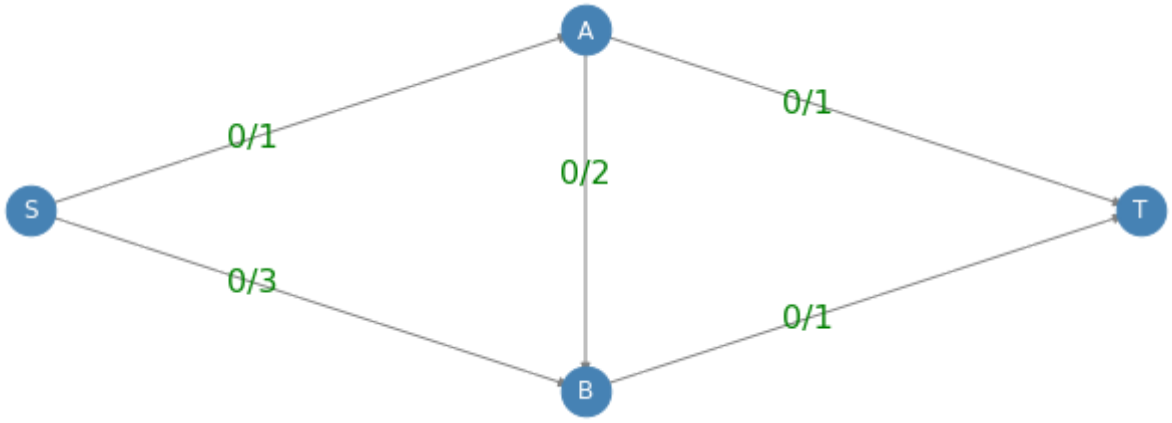

```
Max Flow Graph
```



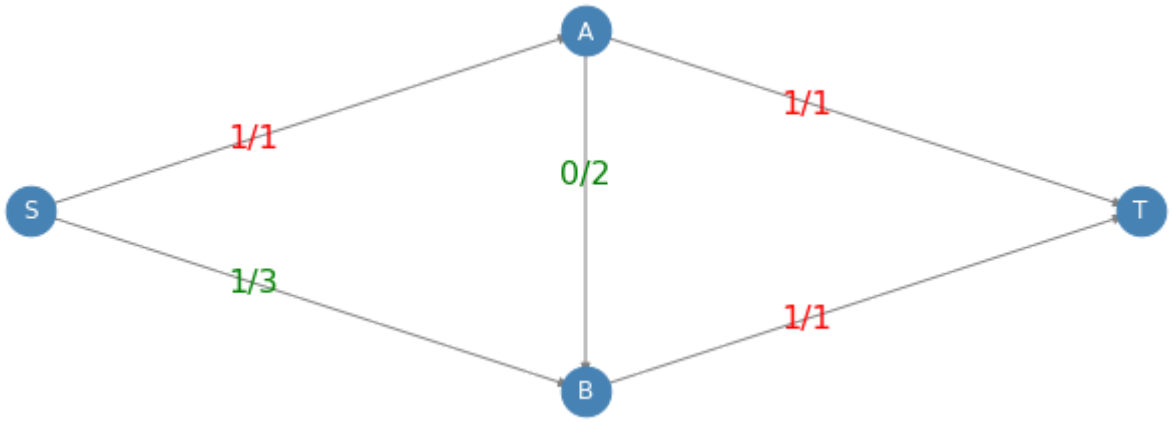

```
Max Flow 15


graph_61 = nx.DiGraph()
graph_61.add_nodes_from('SABT')
graph_61.add_edges_from([
    ('S', 'A', {'capacity': 1, 'flow': 0}),
    ('S', 'B', {'capacity': 3, 'flow': 0}),
    ('A', 'B', {'capacity': 2, 'flow': 0}),
    ('A', 'T', {'capacity': 1, 'flow': 0}),
    ('B', 'T', {'capacity': 1, 'flow': 0})
])

layout2 = {
    'S': [0, 1], 'A': [1, 2], 'B': [1, 0], 'T': [2, 1]
}
```

```
print("Original Graph")
draw_graph(layout2, graph_61)

max_flow_61 = ford_fulkerson(graph_61, 'S', 'T', flow_debug)
print("Max Flow Graph")
draw_graph(layout2, graph_61)
print("Max Flow", max_flow_61)
```

```
Original Graph
```




```
Max Flow Graph
```

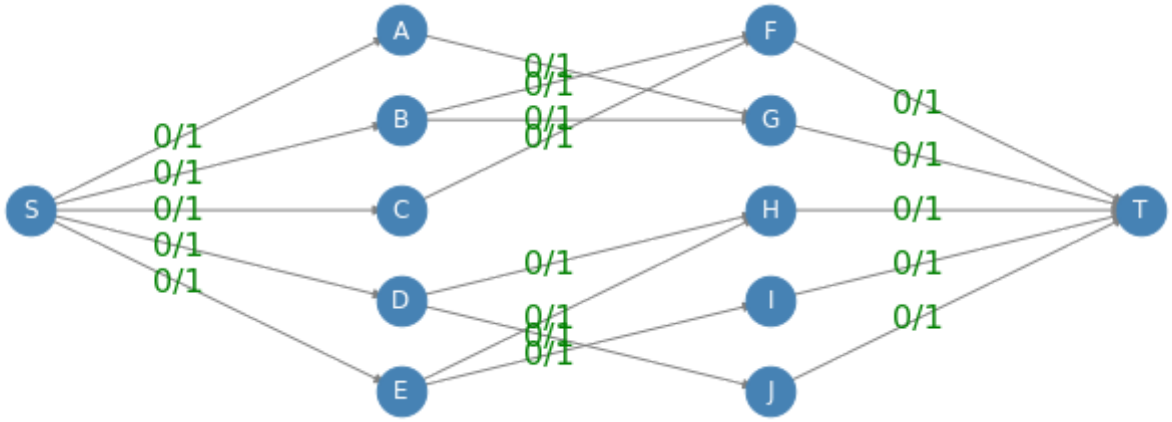



```
Max Flow 2
```

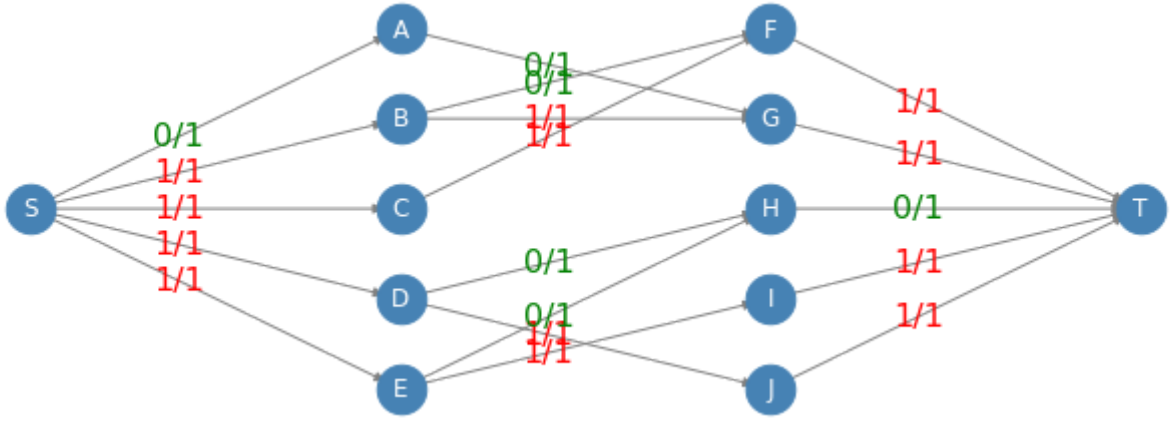
```
graph_63 = nx.DiGraph()
graph_63.add_nodes_from('SABCDEFGHIJT')
graph_63.add_edges_from([
    ('S', 'A', {'capacity': 1, 'flow': 0}),
    ('S', 'B', {'capacity': 1, 'flow': 0}),
    ('S', 'C', {'capacity': 1, 'flow': 0}),
    ('S', 'D', {'capacity': 1, 'flow': 0}),
    ('S', 'E', {'capacity': 1, 'flow': 0}),
    ('F', 'T', {'capacity': 1, 'flow': 0}),
    ('G', 'T', {'capacity': 1, 'flow': 0}),
    ('H', 'T', {'capacity': 1, 'flow': 0}),
    ('I', 'T', {'capacity': 1, 'flow': 0}),
    ('J', 'T', {'capacity': 1, 'flow': 0}),
    ('A', 'G', {'capacity': 1, 'flow': 0}),
    ('B', 'F', {'capacity': 1, 'flow': 0}),
    ('B', 'G', {'capacity': 1, 'flow': 0}),
    ('C', 'F', {'capacity': 1, 'flow': 0}),
    ('D', 'H', {'capacity': 1, 'flow': 0}),
    ('D', 'J', {'capacity': 1, 'flow': 0}),
    ('E', 'H', {'capacity': 1, 'flow': 0}),
```

```
    ('E', 'I', {'capacity': 1, 'flow': 0}),
])

layout3 = {
    'S': [0, 10],
    'A': [1, 20], 'B': [1, 15], 'C': [1, 10], 'D': [1, 5], 'E': [1, 0],
    'F': [2, 20], 'G': [2, 15], 'H': [2, 10], 'I': [2, 5], 'J': [2, 0],
    'T': [3, 10]
}



print("Original Graph")
draw_graph(layout3, graph_63)

max_flow_63 = ford_fulkerson(graph_63, 'S', 'T', flow_debug)
print("Max Flow Graph")
draw_graph(layout3, graph_63)
print("Max Flow", max_flow_63)
```

```
Original Graph
```




```
Max Flow Graph
```




```
Max Flow 4
```

9b

Given a set of n drivers, m riders, and sets of possible riders that each driver can pick up,

i. explain how we can use this maximum-flow algorithm to determine the the maximum number of m

ii. explain how we can additionally extend this to actually find the matchings.

i. We can use the maximum flow algorithm to determine the maximum number of matches by runnin graph where there are edges between a source S and all the drivers with a weight of 1, edges betwee riders with a weight of 1, and edges between the riders and a sink T with a weight of 1, and seeing hc between drivers and riders are saturated. The number of saturated edges is equivalent to the maximu matches.

ii. After running the max flow algorithm, we can trace/follow the backward edges in the residual grapl actually find the matchings.

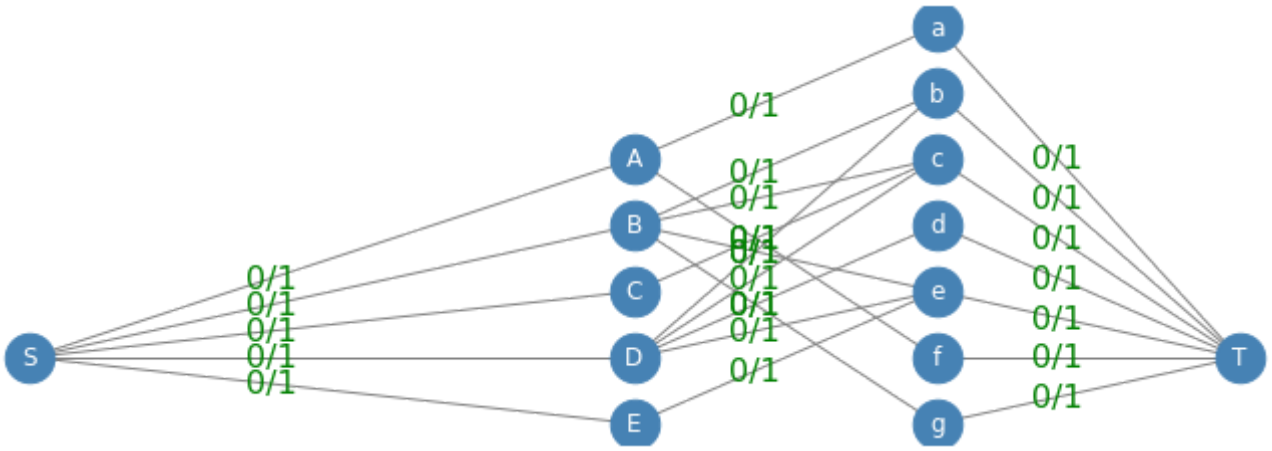
9c

Implement a maximal matching algorithm for Uber drivers and riders. Specifically, given n drivers with specified on m riders, computed the assignments of drivers to riders. Test your algorithm on at least at least 5 riders and drivers each). Explain your examples and your results.

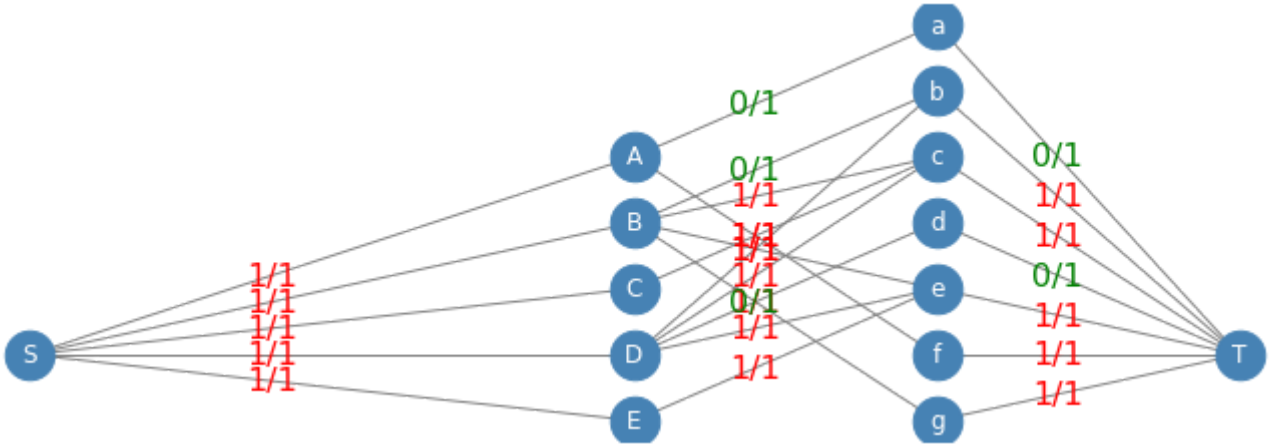
```
#n = 7,
#m = 5
#https://stackoverflow.com/questions/35472402/how-do-display-bipartite-graphs-with-python-n
B = nx.Graph()
B.add_nodes_from('SABCDEabcdefgT')
B.add_edges_from([
    ('A', 'a', {'capacity': 1, 'flow': 0}),
    ('B', 'b', {'capacity': 1, 'flow': 0}),
    ('B', 'c', {'capacity': 1, 'flow': 0}),
    ('B', 'e', {'capacity': 1, 'flow': 0}),

    ('S', 'A', {'capacity': 1, 'flow': 0}),
    ('S', 'B', {'capacity': 1, 'flow': 0}),
    ('S', 'C', {'capacity': 1, 'flow': 0}),
    ('S', 'D', {'capacity': 1, 'flow': 0}),
    ('S', 'E', {'capacity': 1, 'flow': 0}),

    ('a', 'T', {'capacity': 1, 'flow': 0}),
    ('b', 'T', {'capacity': 1, 'flow': 0}),
    ('c', 'T', {'capacity': 1, 'flow': 0}),
    ('d', 'T', {'capacity': 1, 'flow': 0}),
    ('e', 'T', {'capacity': 1, 'flow': 0}),
    ('f', 'T', {'capacity': 1, 'flow': 0}),
    ('g', 'T', {'capacity': 1, 'flow': 0}),

    ('A', 'a', {'capacity': 1, 'flow': 0}),
    ('B', 'b', {'capacity': 1, 'flow': 0}),
    ('B', 'c', {'capacity': 1, 'flow': 0}),
    ('B', 'e', {'capacity': 1, 'flow': 0}),
    ('C', 'c', {'capacity': 1, 'flow': 0}),
    ('D', 'd', {'capacity': 1, 'flow': 0}),
    ('D', 'b', {'capacity': 1, 'flow': 0}),
    ('D', 'c', {'capacity': 1, 'flow': 0}),
    ('D', 'e', {'capacity': 1, 'flow': 0}),
    ('E', 'e', {'capacity': 1, 'flow': 0}),
    ('A', 'f', {'capacity': 1, 'flow': 0}),
    ('B', 'g', {'capacity': 1, 'flow': 0}),
])
```

```
layout4 = {
    'S': [-1, 60], 'A': [1, 150], 'B': [1, 120], 'C': [1, 90], 'D': [1, 60], 'E': [1, 30],
    'a': [2, 210], 'b': [2, 180], 'c': [2, 150], 'd': [2, 120], 'e': [2, 90],
    'f': [2, 60], 'g': [2, 30], 'T': [3, 60]
}


print("Original Graph")
draw_graph(layout4, B)

max_flow_B = ford_fulkerson(B, 'S', 'T', flow_debug)
print("Max Flow Graph")
draw_graph(layout4, B)
print("Max Flow", max_flow_B)
```

```
Original Graph
```




```
Max Flow Graph
```

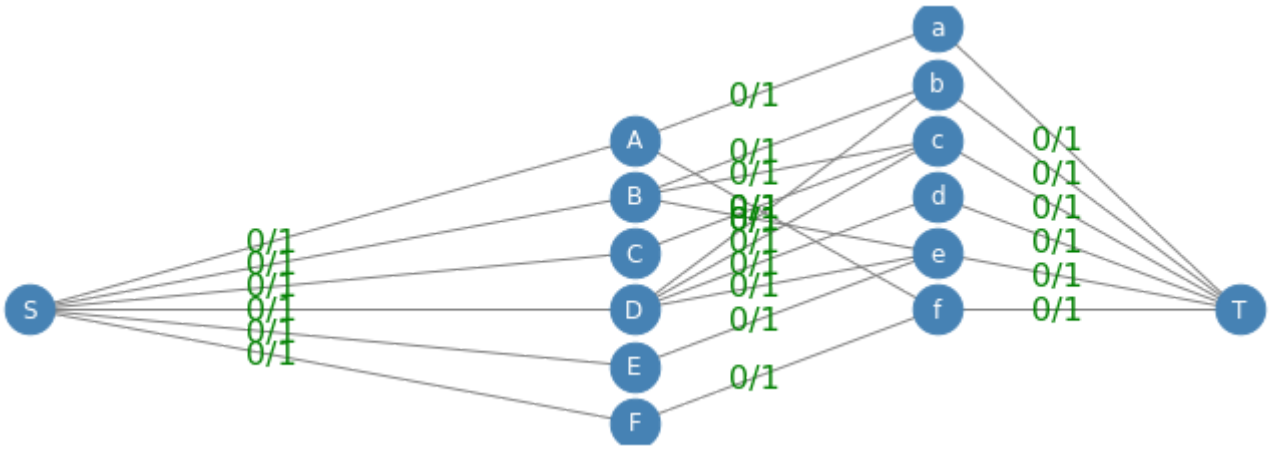



```
Max Flow 5
```

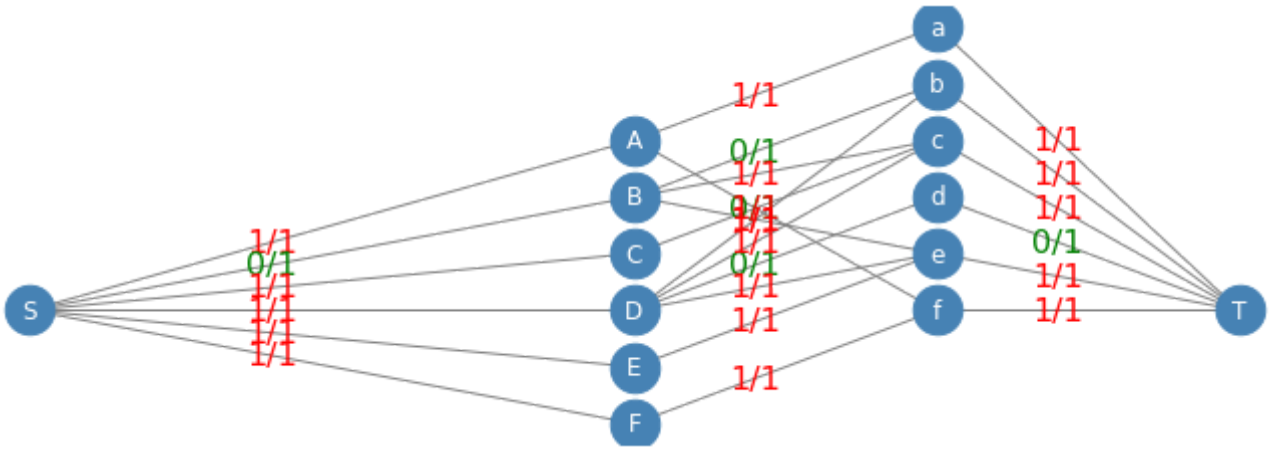
```
#n = 6,
#m = 6

C = nx.Graph()
C.add_nodes_from('SABCDEFabcdefT')
C.add_edges_from([
    ('A', 'a', {'capacity': 1, 'flow': 0}),
    ('B', 'b', {'capacity': 1, 'flow': 0}),
    ('B', 'c', {'capacity': 1, 'flow': 0}),
    ('B', 'e', {'capacity': 1, 'flow': 0}),

    ('S', 'A', {'capacity': 1, 'flow': 0}),
    ('S', 'B', {'capacity': 1, 'flow': 0}),
    ('S', 'C', {'capacity': 1, 'flow': 0}),
```

```
    ('S', 'D', {'capacity': 1, 'flow': 0}),
    ('S', 'E', {'capacity': 1, 'flow': 0}),
    ('S', 'F', {'capacity': 1, 'flow': 0}),

    ('a', 'T', {'capacity': 1, 'flow': 0}),
    ('b', 'T', {'capacity': 1, 'flow': 0}),
    ('c', 'T', {'capacity': 1, 'flow': 0}),
    ('d', 'T', {'capacity': 1, 'flow': 0}),
    ('e', 'T', {'capacity': 1, 'flow': 0}),
    ('f', 'T', {'capacity': 1, 'flow': 0}),

    ('A', 'a', {'capacity': 1, 'flow': 0}),
    ('B', 'b', {'capacity': 1, 'flow': 0}),
    ('B', 'c', {'capacity': 1, 'flow': 0}),
    ('B', 'e', {'capacity': 1, 'flow': 0}),
    ('C', 'c', {'capacity': 1, 'flow': 0}),
    ('D', 'd', {'capacity': 1, 'flow': 0}),
    ('D', 'b', {'capacity': 1, 'flow': 0}),
    ('D', 'c', {'capacity': 1, 'flow': 0}),
    ('D', 'e', {'capacity': 1, 'flow': 0}),
    ('E', 'e', {'capacity': 1, 'flow': 0}),
    ('F', 'f', {'capacity': 1, 'flow': 0}),
    ('A', 'f', {'capacity': 1, 'flow': 0}),
])

layout5 = {
    'S': [-1, 60], 'A': [1, 150], 'B': [1, 120], 'C': [1, 90], 'D': [1, 60],
    'E': [1, 30], 'F': [1, 0],
    'a': [2, 210], 'b': [2, 180], 'c': [2, 150], 'd': [2, 120], 'e': [2, 90],
    'f': [2, 60], 'T': [3, 60]
}


print("Original Graph")
draw_graph(layout5, C)

max_flow_C = ford_fulkerson(C, 'S', 'T', flow_debug)
print("Max Flow Graph")
draw_graph(layout5, C)
print("Max Flow", max_flow_C)
```

```
Original Graph
```




```
Max Flow Graph
```




```
Max Flow 5
```

9d

Now consider the case where there are n drivers and n riders, and the drivers each driver is connecte with probability p. Fix n = 1000 (or maybe 100 if that's too much), and estimate the probability that all matched for varying values of p. Plot your results.

```
diff_p_list = list(np.arange(0, 1.10, 0.10))
prob_all_riders_matched = list()

for p in np.arange(0, 1.10, 0.10):
  brg = bipartite.random_graph(100, 100, p)
  RB_top = set(n for n,d in brg.nodes(data=True) if d['bipartite']==0)
  RB_bottom = set(brg) - RB_top

  brg.add_nodes_from('ST'+str(RB_top)+str(RB_bottom))

  for (u, v) in brg.edges():
    brg.add_edges_from([(u, v, {'capacity': 1, 'flow': 0})])

  for elem in RB_top:
    brg.add_edges_from([('S', elem, {'capacity': 1, 'flow': 0})])

  for elem in RB_bottom:
```

```
      brg.add_edges_from([(elem, 'T', {'capacity': 1, 'flow': 0})])

    max_flow = ford_fulkerson(brg, 'S', 'T', flow_debug)
    prob_all_riders_matched = prob_all_riders_matched + [max_flow / 100]

plt.plot(diff_p_list, prob_all_riders_matched)
plt.xlabel("Probability")
plt.ylabel("Prob all riders matched")
plt.show()
```