CS 5854: Networks, Crowds, and Markets
Homework 3

Instructor: Rafael Pass TAs: Cody Freitag, Drishti Wali

Assigned: October 31, 2019 Due: November 21, 2019, 11:59 pm

Kushal Singh

Collaborators: I worked with Chris Shei and Scooter Blume.
Outside resources: I used the following outside resources: https://www.hbs.edu/faculty/Publication20Files.
Late days: I have used 2 late days on this assignment.

Part 1: Matching Markets and Exchange Networks

1. Consider two sellers, a and b, each offering a distinct house for sale, and a set of two buyers, x and y. x has a value of 2 for a's house and 4 for b's house, while y has a value of 3 for a's house and 6 for b's house, summarized in the following table.

Buyer	a's house	b's house
value for x	2	4
value for y	3	6

   (a) Suppose that a charges a price of 0 for her house, and b charges a price of 1. Is this set of prices market-clearing? As part of your answer, say what the preferred choice graph is with this given set of prices, and use this to justify your answer.

   (b) If the above prices are not market-clearing, compute a set of market-clearing prices p with an associated matching M. Explain how you found them and why they are market-clearing (you don't necessarily need to use Theorem 8.8 in the notes for this, but it could be helpful).

   (c) For the market equilibrium computed above (corresponding to a set of market-clearing prices), compute the social value of the buyers and the social welfare of the buyers and sellers. Briefly explain why these are the maximum possible in this matching market context.

Solution:

   (a) No, this set of prices is not market-clearing. In the preferred choice graph, the utilities for each player are:

$$\textbf{Utility for Buyer } x = \textbf{[2, 4] - [0, 1] = [2, 3]}$$
$$\textbf{Utility for Buyer } y = \textbf{[3, 6] - [0, 1] = [3, 5]}$$

   From this, we can see that both buyer x and buyer y would prefer house B, since this gives each of them the highest (positive utility) with this choice.

   (b) From Theorem 8.8, let's start with all the prices set to zero. From there, there are two possible options. The first is that we observe a perfect matching in the preferred choice graph, in which case we have reached our goal. The second is that a perfect matching does not exist, at which point we would have to first find the constricted set, and then increment by 1 till we reach an equilibrium. Let's begin this process.

   1. Initialize to zero: $a = 0, b = 0$. Then $x \implies a$ with a value of 2, $y \implies a$ with a value of 3, $x \implies b$ with a value of 4, $y \implies b$ with a value of 6. Since $x \implies b$ and $y \implies b$ are both the preferred options, there is no perfect matching and we increment b by 1.

   2. Set new values: $a = 0, b = 1$. Then $x \implies a$ with a value of 2, $y \implies a$ with

a value of 3, $x \implies b$ with a value of 3, $y \implies b$ with a value of 5. Since $x \implies b$ and $y \implies b$ are still both the preferred options, there is no perfect matching and we increment b by 1 once again.

3. Set new values: $a = 0, b = 2$. Then $x \implies a$ with a value of 2, $y \implies a$ with a value of 3, $x \implies b$ with a value of 2, $y \implies b$ with a value of 4. Since $y \implies b = 4$ is greater than $y \implies a = 3$, and $x \implies a = x \implies b = 2$ (i.e. x prefers both a and b equally, we have found our perfect matching.

x would purchase a for 0, with a utility of 2
y would purchase b for 2, with a utility of 4

(c) For the market equilibrium computed above, the maximum possible social value = 8. We can compute this by looking at the best possible values for x, y choosing either a or b (graphically, this means finding the max of the summation of the values along the main diagonals). In other words:

$max(x \implies a + y \implies b, x \implies b + y \implies a)$
$= max(2 + 6, 4 + 3)$
$= max(8, 7)$
$= 8$, when x chooses a, y chooses b.

Comparing this with the prices from above, we can see that:

a sells at 0, with utility = **0**.
b sells at 2, with utility = **2**.
x buys from a at a price of 0, with utility = **2**.
y buys from b at a price of 2, with utility $6 - 2 =$ **4**.

Summing the above utilities, we get $0 + 2 + 2 + 4 = 8$, which corroborates our finding from above that the maximum possible utility = 8.

□

2. Suppose now we have a set of three sellers a, b, and c, and a three buyers labeled, x, y, and z. The valuations of the buyers are summarized in the following table.

Buyer	a's house	b's house	c's house
value for x	5	7	1
value for y	2	3	1
value for z	5	4	4

(a) Using the algorithm from Theorem 8.8 in the notes, compute a market equilibrium (consisting of a matching and set of prices). Show your work.

(b) Recall that in chapter 9, we showed that every matching market context can be encoded as an exchange network, and furthermore, exchange networks are "asymmetric" in the sense that the buyers and sellers have the same role. As a result, we can flip the script such that the sellers are now the "buyers" and the buyers are now the "sellers." The corresponding matching market context now has the following set of values.

Seller	if x buys	if y buys	if z buys
value possible for a	5	2	5
value possible for b	7	3	4
value possible for c	1	1	4

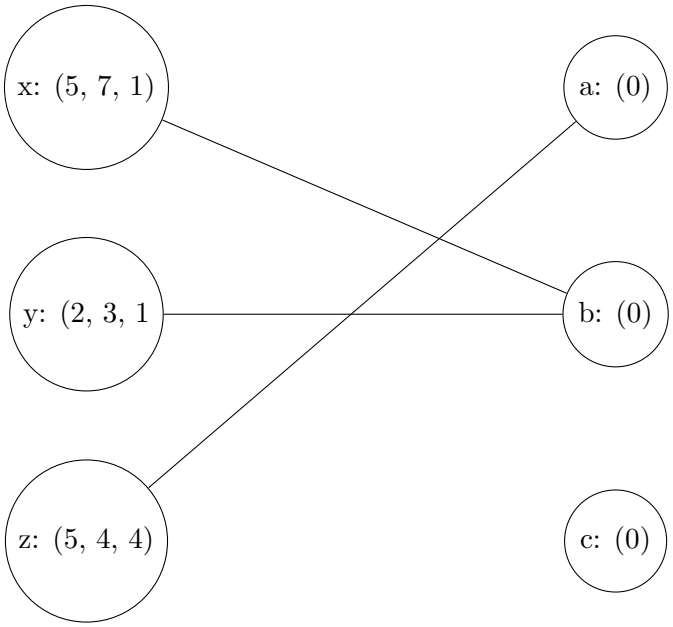
You can think of the value of a seller being the maximum value it can charge for a home. Now the prices for the buyer correspond to discounts on the house (increasing value for the buyers).

Using the algorithm from Theorem 8.8 in the notes, compute a market equilibrium (consisting of a matching and set of "prices") for this new matching market context. Show your work.

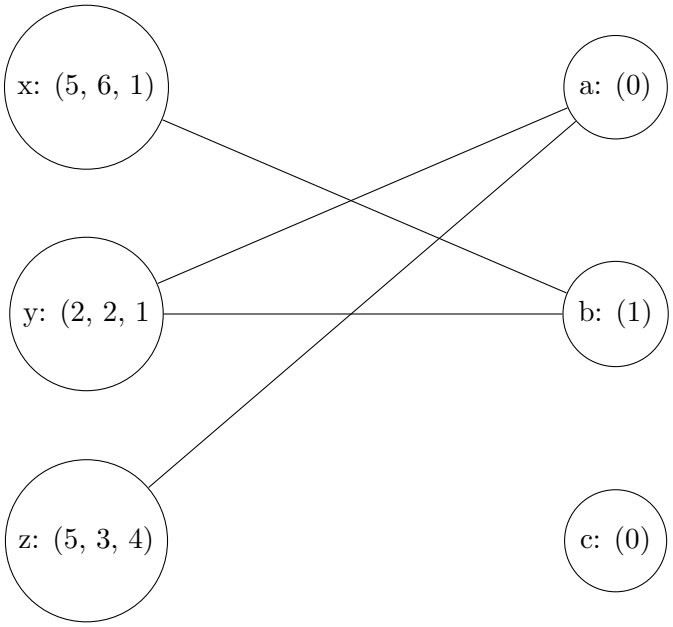
(c) Interpret the matching and "prices" computed in part (b) as a matching and actual prices charged when selling the houses. How do these values compare to those computed in part (a)? Give a 1-3 sentence intuitive explanation of why the values might be different. In particular, explain what the consequence is that the algorithm of Theorem 8.8 always has a "free" item in the resulting market equilibrium.

Solution:

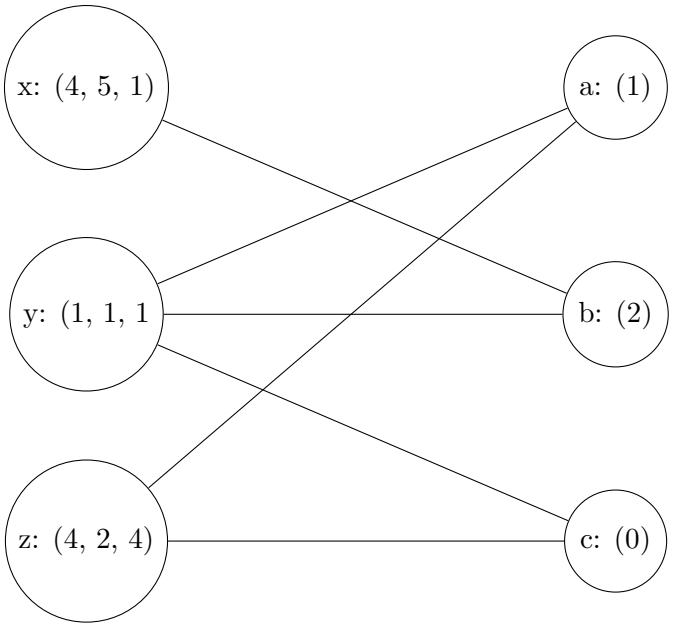
(a) To compute a market equilibrium using the algorithm from Theorem 8.8 in the notes, we first initialize everything at 0, and then create the preferred matching graph.



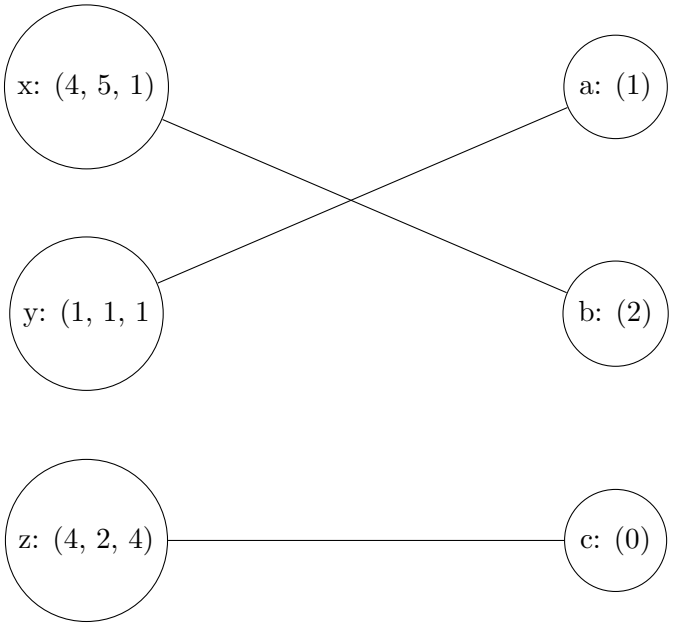
We then increment the constricted set $\{b\}$ by 1, and recreate the preferred matching graph.



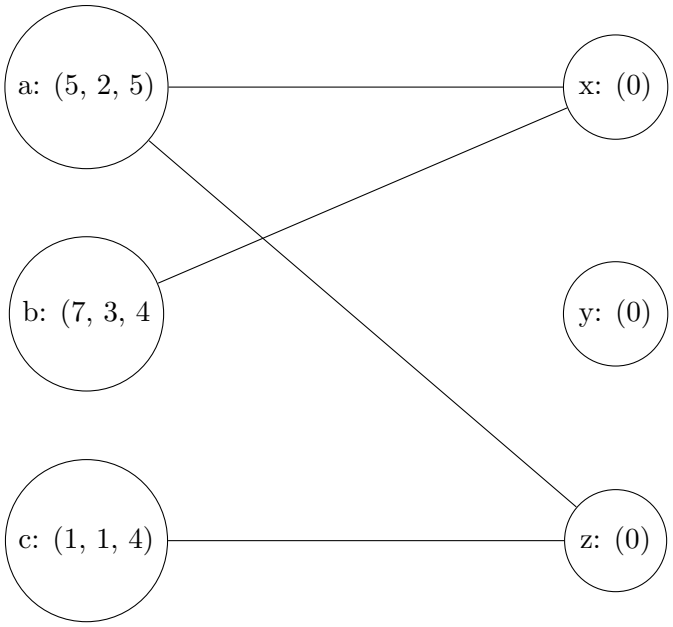
We then increment the constricted set $\{a, b\}$ by 1, and recreate the preferred matching graph.



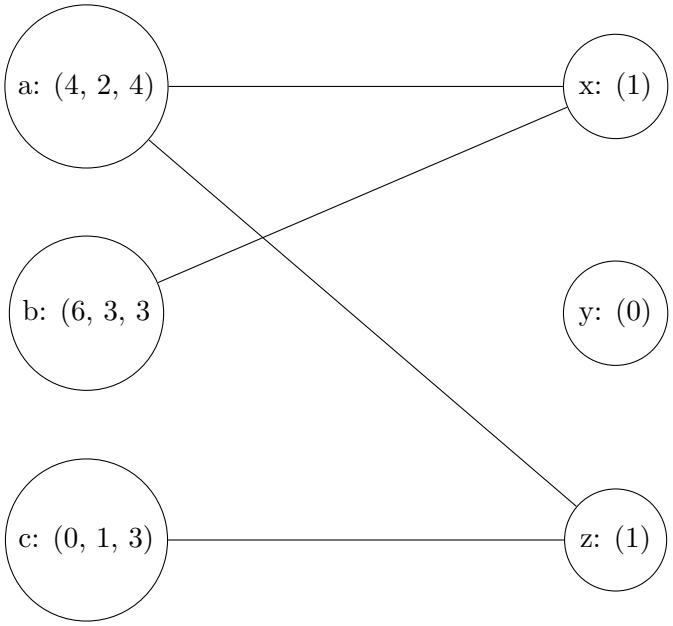
From this, we can observe that a perfect matching does exist, with $x(5) \implies b(2)$, $y(1) \implies a(1)$, $z(4) \implies c(0)$.



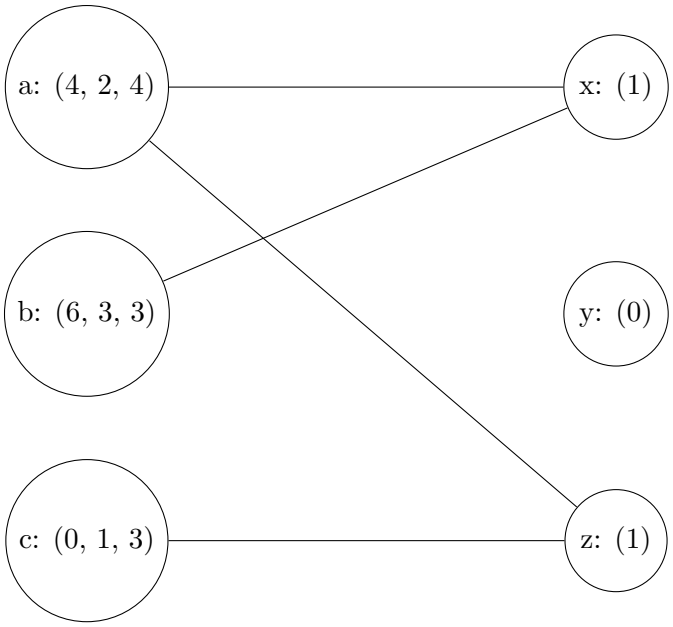
(b) To compute a market equilibrium for this new matching market context using the algorithm from Theorem 8.8, we first initialize everything at 0, and then create the preferred matching graph.



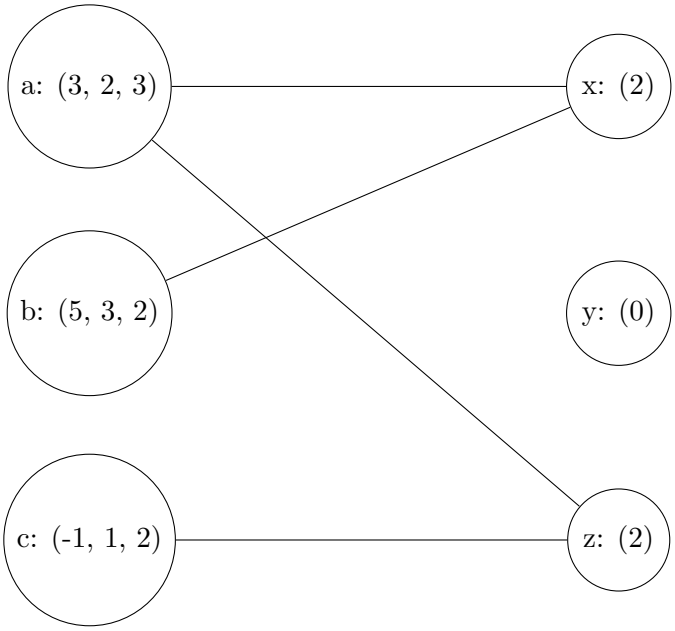
We then increment the constricted set $\{x, z\}$ by 1, and recreate the preferred matching graph.



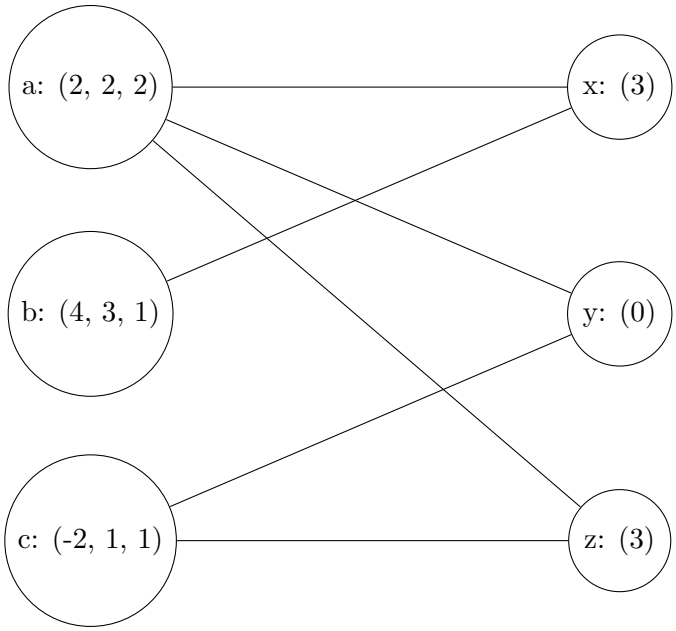
We then increment the constricted set $\{x, z\}$ by 1, and recreate the preferred matching graph.



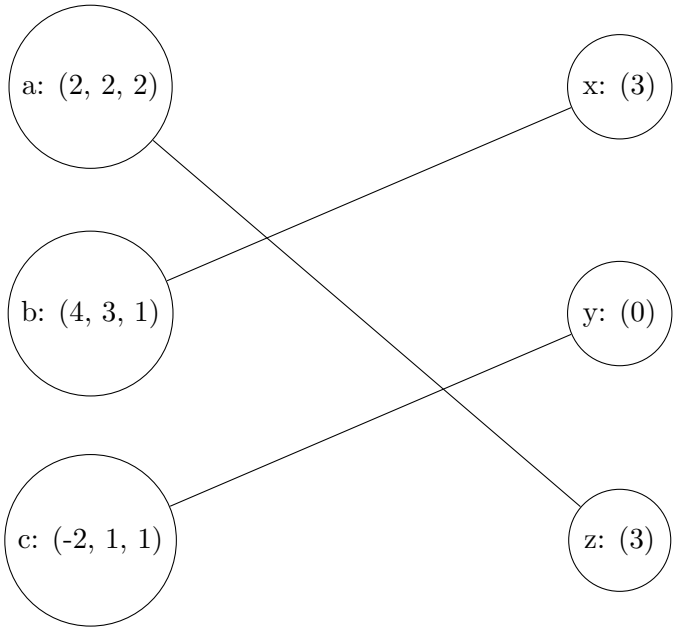
We then increment the constricted set $\{x, z\}$ by 1, and recreate the preferred matching graph.



We then increment the constricted set $\{x, z\}$ by 1, and recreate the preferred matching graph.



From this, we can observe that a perfect matching does exist, with $a(2) \implies z(3)$, $b(4) \implies x(3)$, $c(1) \implies y(1)$.



(c) In part (a), we have the following transactions:
x buys from b at a price of 2, with utility = **2**.
y buys from a at a price of 1, with utility $6 - 2 =$ **4**.
z buys from c at a price of 0, with utility $6 - 2 =$ **4**.

Summing the above utilities, we get $U(houses) = 2 + 1 + 0 = 3$, $U(buyers) = 5 + 1 + 4 = 10$, and so the total social value $= U(houses) + U(buyers) = 13$.

In part (b), we have the following transactions:
a sells to z at a price of 2.
b sells to x at a price of 4.
c sells to y at a price of 1.

Summing the above utilities, we get $U(houses) = 2 + 4 + 1 = 7$, $U(buyers) = 3 + 0 + 3 = 6$, and so the total social value $= U(houses) + U(buyers) = 13$.

From these results, we can observe that selling prices are higher in part (b) (seller discounts). Buyers pay less when houses compete against one another than when buyers themselves compete against one another.

When we start with the prices initialized at 0, we gradually increase the prices until we reach the minimum selling price for a market equilibrium.

When we start with the discounts initialized at 0, we gradually increase the discounts until we reach the minimum discount (or, maximum selling price) for a market equilibrium.

The consequence of always having something priced at 0 means that once we reach market equilibrium, someone will always get something for 'free', or for zero discount.

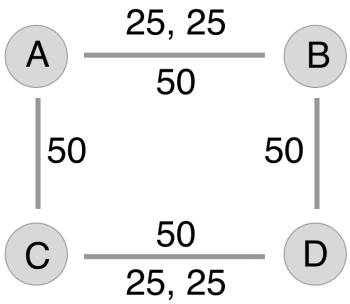
□

3. (a) We observed (Claim 9.5) that an exchange network consisting of a cyclic graph of 3 nodes has no stable outcome. Does this generalize to every cyclic graph? If not, can we characterize for which values of n the n-node cyclic graph has a stable outcome? Justify your answers.

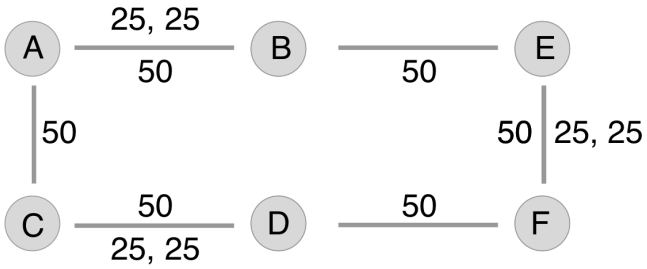
   (b) Show by constructing an example that an exchange network that contains a 3-node cycle (but doesn't necessarily entirely consist of one) can still have a stable outcome.

Solution:

(a) Even though a cyclic graph of just 3 nodes will never have a stable outcome, this fact does NOT generalize for every cyclic graph. It only generalizes for n-node cyclic graphs which have odd values of n. In other words, if an n-node cyclic graph contains an even number of nodes, then we can find a stable outcome. Below is an example where we can find an equilibrium in a 4-node graph.



If we add two nodes to the previously generated even-node cyclic graph, then we can continue to maintain an equilibrium, by pairing the two new nodes with one another.

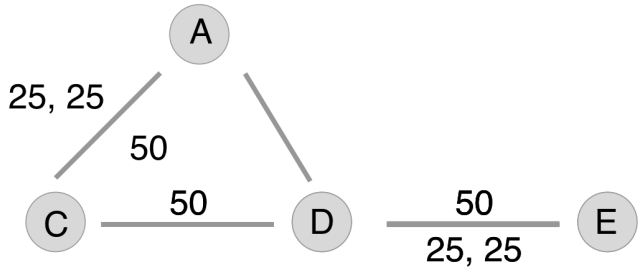


This would be our inductive step. Then, by induction, we can continually add two nodes to the graph and pair them with each other, leading to our conclusion that an equilibrium can be found for every even n-node cyclic graph.

For further justification as to why this doesn't work for the odd-node cyclic graph, we can observe that there will always be one node that is not paired. Let's call the unpaired node $x_{unpaired}$. From theorem 9.2 in the notes, we know that if $a(x) + a(y) < v(x, y)$, then we do

not have a stable outcome. Since $x_{unpaired}$ is not paired with any other node, we know that $a(x) = 0$. Furthermore, since $a(y) < v(x, y)$, the above equation holds true. Therefore, for every odd n-node cyclic graph, a stable outcome cannot be found.

(b) Intuitively, to construct a 3-node cycle with a stable outcome, we would need to add at least another node jutting off to the side in order to balance the graph out.



□

Part 2: Auctions and Mechanism Design

4. Consider an auction where a seller wants to sell one unit of a good to a group of bidders. The seller runs a sealed-bid, second-price auction. Your firm will bid in the auction, but it does not know for sure how many other bidders will participate in the auction. There will be either two or three other bidders in addition to your firm. All bidders have independent, private values for the good. Your firm's value for the good is c. What bid should your firm submit, and how does it depend on the number of other bidders who show up? Give a brief (1-3 sentence) explanation for your answer.

   Solution: Your firm should bid c, regardless of the number of other bidders that show up, since bidding truthfully always provides a non-negative utility. To justify, we have two scenarios to consider.

   The first scenario is when your firm bids c, and there exists exactly one other bidder who bids $c' > c$. In this case, your current utility is zero. Any deviation to a price less than c' will still lead to a utility of zero, so you don't have an incentive to change your action profile. Deviating to a price greater than c' would mean that your utility $= c - c' < 0$, which means you were worse off than when you started off, so your incentive to deviate is very low.

   The second scenario is when the other bidder bids at a price $c' < c$. In this case, your current utility is $c - c' > 0$. Any deviation to a price $> c'$ would maintain the same utility, so you don't have an incentive to change your action profile. Deviating to a price $< c'$ would yield a utility of zero, which means you were worse off than when you started off, so your incentive to deviate is very low.

   Furthermore, since the only bids that matter in a sealed-bid second-price auctions are the top two bids, the number of other bidders who show up does not matter, which means the rationale above applies, regardless of the number of bidders. □

5. Consider a second-price, sealed-bid auction with two bidders who have independent, private values v_i which are either 1 or 3. For each bidder, suppose the probabilities of v_i being 1 or 3 are independent (from other bidders) and both 1/2. (If there is a tie at a bid of x for the highest bid the winner is selected at random from among the highest bidders and the price is x.)

   (a) What is the seller's expected revenue from the auction? Justify your answer.

   (b) Assume that we add a third bidder that behaves identically to the first two (whose value v_i is also chosen independently of those for the first two). What is the seller's expected revenue? Justify your answer.

   (c) Explain the trend you noticed between parts (a) and (b)—that is, why changing the number of bidders affects (or doesn't affect) the seller's expected revenue.

Solution:

(a) Let's first consider the probabilities of all the possible cases arising from this game:

$$\Pr(v_i = 1, v_2 = 1) = \Pr(v_i = 1, v_2 = 3) = \Pr(v_i = 3, v_2 = 1) = \Pr(v_i = 3, v_2 = 3) = \frac{1}{4}$$

Therefore, the seller's expected revenue is $R = \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 3 = \frac{6}{4} = \frac{3}{2} = 1.5$.

(b) Let's first consider the probabilities of all the possible cases arising from this game:

$$\Pr(v_1, v_2, v_3) = \frac{1}{8}, \forall (v_1, v_2, v_3) \in \{1, 3\}$$

Therefore, the seller's expected revenue is $R = \frac{1}{8} * (1+1+1+1+3+3+3+3) = \frac{16}{8} = 2$.

(c) In this setup, it appears as though the seller's expected value increases as the number of bidders increases. From the above computations, we can infer a generalization for n players:

$$R = \frac{1}{2^n} \cdot [1 + n \cdot 1 + 3 \cdot (2^n - n - 1)]$$

When two or more bidders bid 3, the seller will make an expected revenue of 3. In all other cases, the expected revenue is 1. As the number of bidders approaches infinity, it is more likely that at least two bidders will bid 3. If we let n = number of bidders, then:

Pr(no one bidding three) : $1/2^n$
Pr(one person bidding three) : $n/2^n$

Likelihood of $R(1) = (n+1)/2^n$
Likelihood of $R(3)$ = 1 - likelihood of $R(1)$

Expected revenue = Likelihood of $R(1)$ * 1 + Likelihood of $R(3)$ * 3.
As n approaches infinity, the likelihood of $R(1)$ approaches zero, and the likelihood of $R(3)$ approaches 1. This yields an expected revenue = $(0 * 1 + 1 * 3) = 3$.

□

* **Bonus Question 1.** Let's say we define a "third-price auction" in the same manner that we defined the first-price and second-price auctions. Is this mechanism DST, NT, or neither? Does it DST-implement, NT-implement, Nash-implement, or fail to implement social welfare maximization? Justify your answers, by proof or by counterexample.

Solution: Let's consider player p with valuation v, and 2 other players with valuations v' and v'' such that $v < v'' < v'$. Assuming all other players bid their true valuation, the pth player can benefit from bidding above the true valuation in the following manner.

Bid 1: Bid of the pth player (not equal to true value).
Bid 2: v' (highest valuation amongst all players); v : (true valuation of the ith player)
Bid 3: v'' (second highest valuation amongst all players)

As long as p bids below v', i's utility remains 0 (since he/she loses). However, if p bids above v', he/she wins the auction and receives the item for a price v'' (3rd highest bid) $< v$. Clearly, since this player has incentive to lie, the auction is neither DST nor NT. □

Part 3: Sponsored Search

6. Suppose a search engine has two ad slots that it can sell. Slot a has a clickthrough rate of 10 and slot b has a clickthrough rate of 5. There are three advertisers who are interested in these slots. Advertiser x values clicks at 3 per click, advertiser y values clicks at 2 per click, and advertiser z values clicks at 1 per click.

   Compute the socially optimal allocation and the VCG prices for it (using the Clarke pivot rule). Show your work in full, and explain what your answer means in the sponsored search context.

Solution: The socially optimal allocation would be to provide the most clicks to the advertiser who values it the most, and the second largest clicks to the advertiser who values it second most. That being said, in this scenario, we would provide slot a to x, and slot b to y. Using the Clarke Pivot Rule to calculate the VCG prices, we want each advertiser to pay the externality it has caused on the system.

Advertiser x should pay 15
Social Value when x is present: $(5 * 2) = 10$
Social Value when x is not present: $(10 * 2 + 5 * 1) = 25$
Price to pay: $25 - 10 = 15$

Advertiser y should pay 5
Social Value when y is present: $(10 * 3) = 30$
Social Value when y is not present: $(10 * 3 + 5 * 1) = 35$
Price to pay: $35 - 30 = 5$

Advertiser z should pay 0
Social Value when z is present: $(10 * 3 + 5 * 2) = 40$
Social Value when z is not present: $(10 * 3 + 5 * 2) = 40$
Price to pay: $40 - 40 = 0$

From this, we can see that x pays 15 for slot a, y pays 5 for slot b, and z does not get a slot.

□

Part 4: Implementing Matching Market Pricing

The goal of this exercise is to implement an algorithm for finding market-clearing and VCG prices in a bipartite matching market. Include your code in matching_market.py.

7. Recall the procedure constructed in Theorem 8.8 of the notes to find a market equilibrium in a matching market.

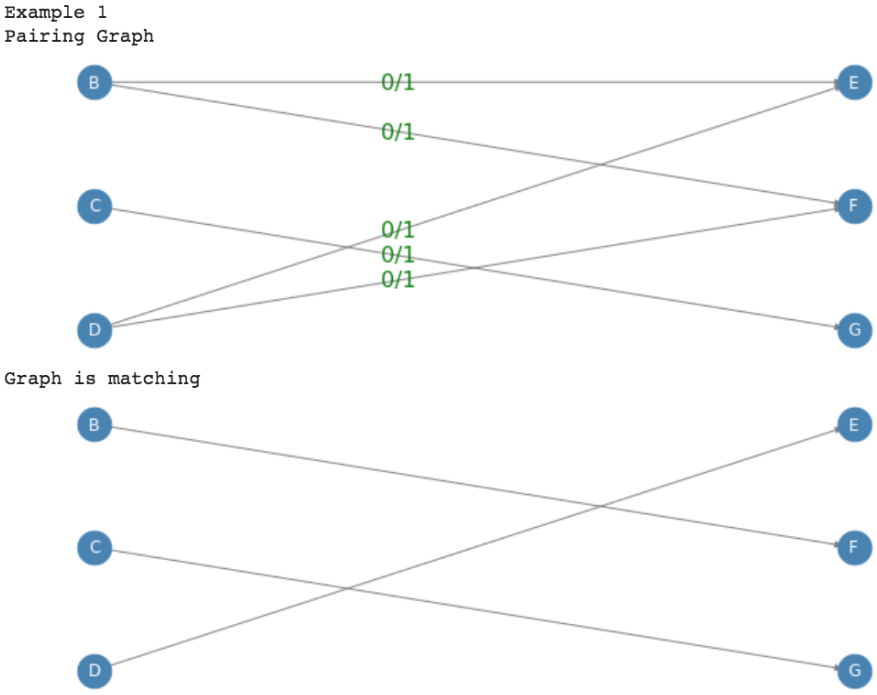
   (a) The first step of this procedure involves either finding a perfect matching or a constricted set. Recall that this can be done using maximum flow. Now, using your maximum-flow implementation from assignment 2, implement an algorithm that finds either a perfect matching M or a constricted set S in a bipartite graph.

   (b) Now, given a bipartite matching frame with n players, n items, and values of each player for each item, implement the full procedure to find a market equilibrium.

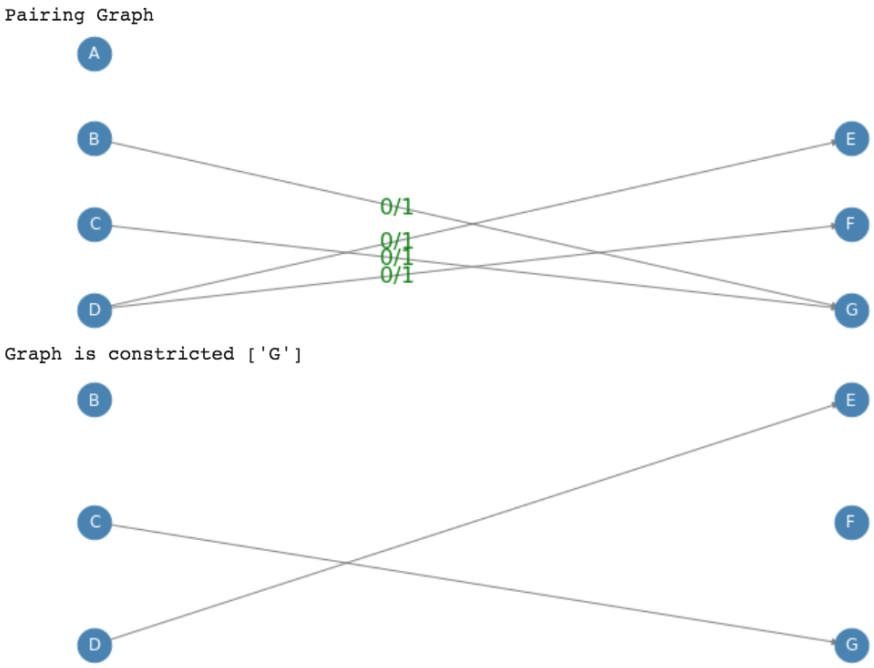
   (c) Submit your code along with its output on the matching market frame in figure 8.3 as well as 3 other small (10-20 node) test examples of your choice. Include the input and outputs in a file called p7.txt.

Solution:

(a) Finding perfect matching:

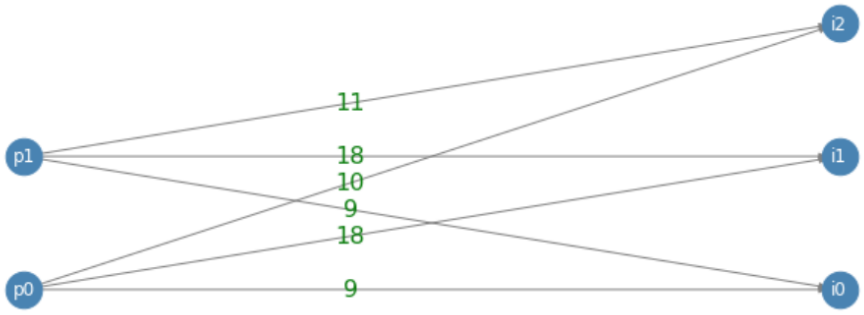


Finding constricted set G (B, C both want to pair with G):

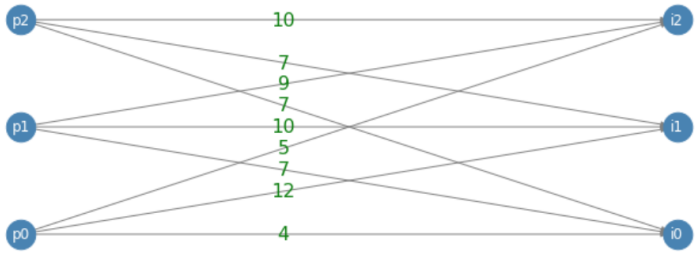


(b) *****SEE CODE*****

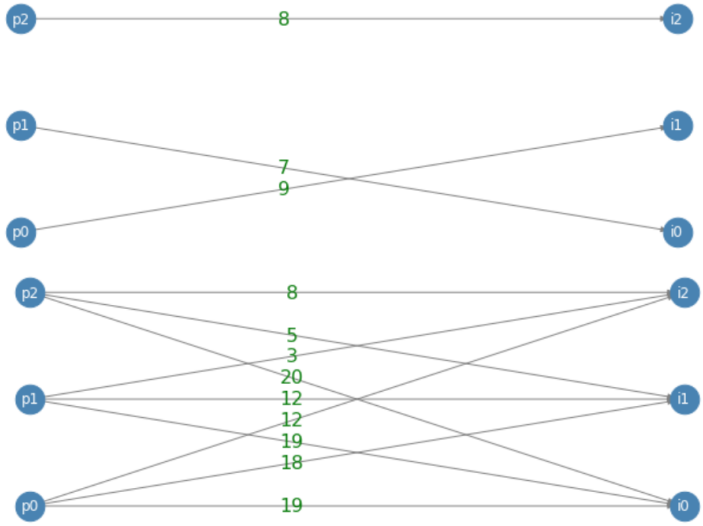
(c) **TEST EXAMPLES**



Example input

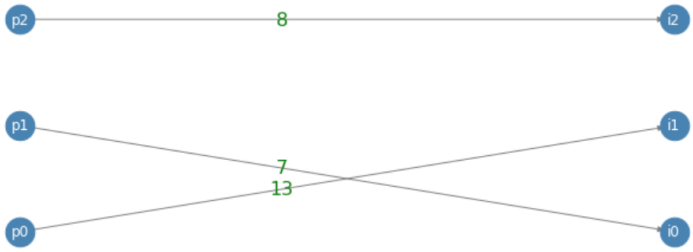


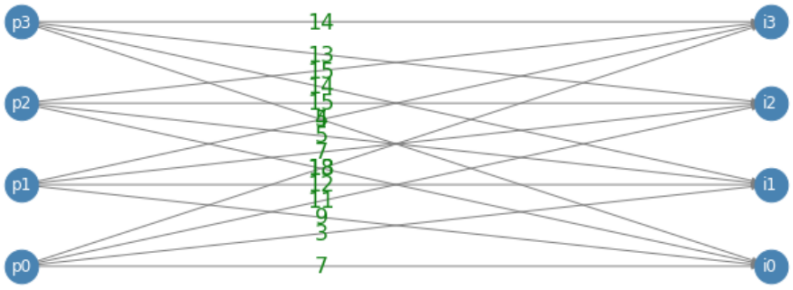
Example output equilibrium
[('p0', 'i1', {'weight': 9}), ('p1', 'i0', {'weight': 7}), ('p2', 'i2', {'weight': 8})]
prices (i0,i1,...in): [0. 3. 2.]



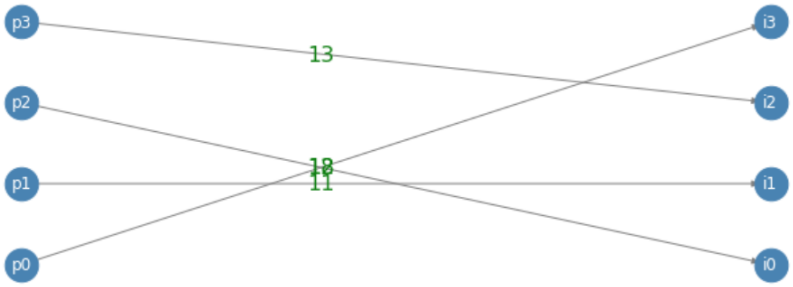
Example output equilibrium
[('p0', 'i1', {'weight': 13}), ('p1', 'i0', {'weight': 7}), ('p2', 'i2', {'weight': 8})]
prices (i0,i1,...in): [12. 5. 0.]

i1






```
Example output equilibrium
[('p0', 'i3', {'weight': 12}), ('p1', 'i1', {'weight': 11}), ('p2', 'i0', {'wei
 prices (i0,i1,...in):  [0. 1. 0. 1.]
```



□

8. Now, given a matching market frame, we will implement VCG pricing in this frame according to the results of Theorem 10.8 in the notes.

   (a) In order to do this, we must find the socially optimal outcome. Briefly justify that the outcome that your algorithm from the previous question finds is socially optimal (this can be done by simply stating 1-2 theorems from the notes).

   (b) Finally, implement the VCG mechanism and Clarke pivot rule to construct an algorithm that produces a positive set of VCG prices in any matching market frame.

   (c) Submit your code along with its output on the matching market frame in figure 8.3 as well as 3 other small (10-20 node) test examples of your choice. Include the input and outputs in a file called p8.txt.

Solution:

(a) From Theorem 8.7, we know that a market equilibrium optimizes social welfare, and we use Corollary 8.6 for the proof.

Proof for Theorem 8.7: Since a market equilibrium is a perfect matching in the preferred choice graph, every buyer receives an item that maximizes their utility. In addition, the utilities of the sellers are maximized for the fixed set of prices. Following from corollary 8.6, our equilibrium maximizes social value.

(b) *****SEE CODE*****.

(c) *****SEE CODE AND TEXT FILES*****.

□

9. Now simulate your VCG pricing algorithm in the following context.

   (a) First, construct a graph of 20 buyers and 20 items. Assume that "item" i is actually a bundle of i identical goods. Now assign each player a random value per good (say, from 1 to 50; ties can be allowed). You shouldn't need to do any additional work on your algorithm to do this, instead just set each buyer's value per bundle appropriately. How should we set these values?

   (b) Having set the values accordingly, run your VCG pricing algorithm and turn in the results in a file called p9.txt. Explain why the results you obtained make sense in the context above.

Solution:

(a) We have 20 players with random values between 1 and 50. We also have 20 bundles with varying items from 1 to 20. Edge weights are set by valuation $\times$ number of items in bundle.

(b) Example: 20 players and 20 bundles of identical items

```
[49, 47, 38, 35, 32, 32, 32, 32, 29, 25, 22, 21, 15, 14, 7, 7, 3, 2, 2, 2]
[20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1]
```

Results:

```
[[980, 931, 882, 833, 784, 73
[('p0', 'i0', {'weight': 980}
Player p0's externality: 397
Player p1's externality: 350
Player p2's externality: 312
Player p3's externality: 277
Player p4's externality: 149
Player p5's externality: 181
Player p6's externality: 213
Player p7's externality: 245
Player p8's externality: 120
Player p9's externality: 95
Player p10's externality: 73
Player p11's externality: 52
Player p12's externality: 37
Player p13's externality: 23
Player p14's externality: 9
Player p15's externality: 16
Player p16's externality: 6
Player p17's externality: 0
Player p18's externality: 2
Player p19's externality: 4
```

In general, these results make sense. Players who value an item the most should have the largest externality. In our example, p_0 values items at 49 and, therefore, has the largest externality. Similarly, those who value an item the least should have the smallest externality. However, since ties can occur, multiple players can have the same valuation of an item. This can cause one of the players to pay less externality than the other. For instance, players p_{17}, p_{18}, p_{19} all have valuations of 2. This means that p_{17} does not pay anything, p_{18} pays the externality of $p_{17} = 2 * 1 = 2$, p_{19} pays the externality of both p_{17} and $p_{18} = (2 \times 2 + 2 \times 1)(2 \times 1) = 4$.

□

* **Bonus Question 2.** (Please note that each part of this question is intended to be harder than the previous parts; each part will be graded separately, and you are not required to do the whole thing.)

(a) Implement an algorithm for GSP pricing in a matching-market context. Compare your VCG prices from the previous question to the GSP prices when run in the same contexts (with the same randomness). Also, try to find and characterize some contexts where VCG and GSP prices are similar, and where they are wildly different. Include a file called bonus-2a.txt displaying input and output on a few examples.

(b) GSP isn't truthful, so interesting things might happen if we run BRD on a GSP matching market auction (where a player's "strategy" is their valuation report). Implement an algorithm that picks a random starting point and runs BRD to attempt to find a GSP equilibrium state. What happens? Does BRD converge; if so, how quickly? Include a file called bonus-2b.txt displaying input and output on a few examples.

(c) *(Very difficult.)* Either prove that BRD will converge in a GSP context, or disprove it by counterexample.

Solution:

(a)

(b)

(c)

□

Part 5: Exchange Networks for Uber

We will construct a simplified market scenario for a ridesharing app like Uber. Our world will consist of an $n \times n$ grid (think of 100×100 as a test example), and there will be two types of participants, riders and drivers.

- A rider R is specified by a current location $(x_0, y_0) \in [n] \times [n]$, a desired destination $(x_1, y_1) \in [n] \times [n]$, and a value for reaching that destination.
- A driver D is specified by a current location $(x_0, y_0) \in [n] \times [n]$.

We define the cost of a matching between a rider R and a driver D, $c(R, D)$, to be the distance from the driver to the rider and then to the destination (measured via manhattan distance, so the distance from $(0, 0)$ to $(5, 2)$ is 7).

10. Encode the above example as an exchange network and implement it in 'uber.py.' Namely, define a graph $G = (V, E)$ where the vertices are the riders and drivers and there is an edge between every rider and driver. What value should we associate with each edge in the graph? Justify your answer.

11. Using your algorithm for matching markets above, implement a procedure (in uber.py) that computes a stable outcome in this context. Note that there may not be the same number of riders and drivers.

    (a) Construct at least two test examples with at least 5 riders and drivers. Discuss what the stable outcome is for your chosen examples. Explain what your results say about how much the riders are charged and how much the drivers are profiting.

    (b) Implement a procedure that on a 100×100 grid (1) generates r riders (located randomly in the grid) each with a value of 100, d drivers (also located randomly in the grid) and (2) computes a stable outcome given this context. Run your procedure many times when $r = d$ (say, 10 each), r is much less than d (say, 5 vs. 20), and when r is much greater than d (say, 20 vs. 5). Discuss your results in detail. Specifically, discuss prices and profits for different ranges of r and d.

12. With ridesharing apps, drivers often have preferences for where they want to drive. For example, they know they are more likely to get a high value ride from the airport. Describe in a few sentences how you you could modify the existing setup to include driver's preferences.

Solution:

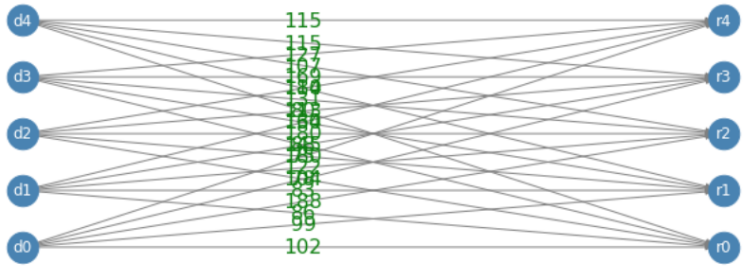
10. The edge weights in our graph = value - cost. This represents the potential benefit between a rider and a driver. If cost is high (i.e. far apart, which means low benefit between the two), then they would choose not to be paired. If the cost is low (i.e. they are close), then it's a good match. We defined value - cost because that is in the definition of a matching market.

Following chapter 8.1: Given a set of players (buyers) X = [n] = {1, . . . , n} and a set of n items Y :

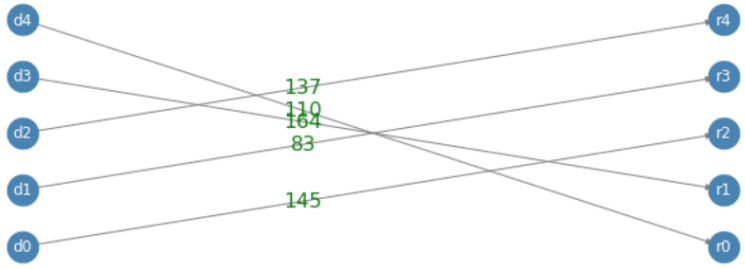
- we associate with each player (buyer) i ∈ X, a valuation function vi :Y → N. For each y ∈ Y , vi(y) determines i's value for item y.
- we associate with each item y ∈ Y , a price p(y) ∈ N.
- a buyer i who receives an item y gets utility vi(y) − p(y).

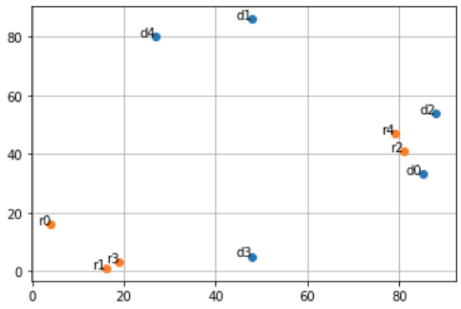
11. (a) TEST EXAMPLES

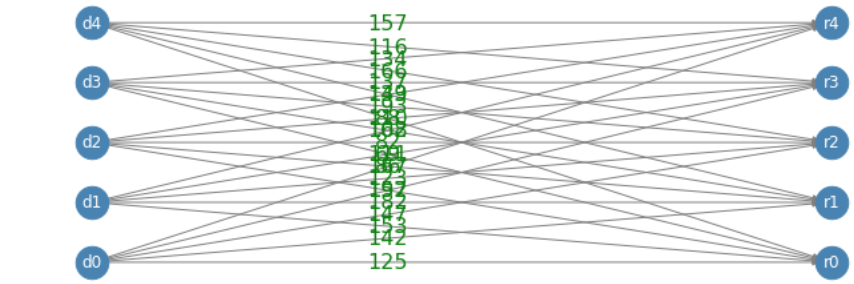
```
[('d0', 'r0', {'weight': 102}), ('d0', 'r1', {'weight': 99}), ('d0', 'r2', {'weig
```



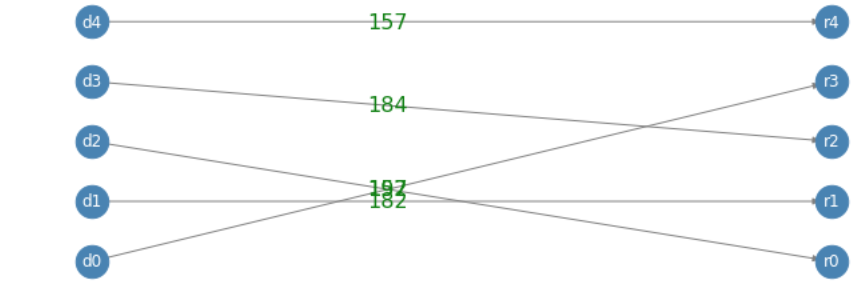

```
Driver discounts [d0,d1,...dn]:  [43, 5, 47, 0, 3]
Drivers recieve [d0,d1,...dn]:  [145, 83, 137, 164, 110]
```

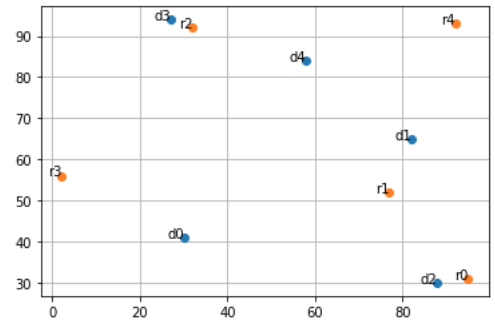







```
Driver discounts [d0,d1,...dn]:  [0, 0, 0, 9, 0]
Drivers recieve [d0,d1,...dn]:  [157, 182, 192, 184, 157]
```





In the graphs above, **Driver discounts** represent the utility that each rider gets, and **Drivers receive** represent the utility each driver gets. **Drivers receive** + **Driver discounts** = weight of edge = potential utility of that specific partnership.

Drivers receive a larger amount of the potential utility. In our first example, the two drivers that gave the largest discounts (40) were d_0 and d_2, who received 75% of potential utility. This was probably because both of them were close to r_4 and r_2, so they were competing with one another.

In our second figure, d_4 is the only driver which gave a discount. The one driver and rider are close, so the riders don't have many driver options. d_4 probably had to give a discount (5%) because it was competing with d_1 for r_4.

(b) We implemented a way to generate maps on a 100×100 grid and assign values that riders are willing to pay. We decided to use 200 nodes total because we wanted to be able to keep track of dummy riders and mismatches. In other words, if a driver

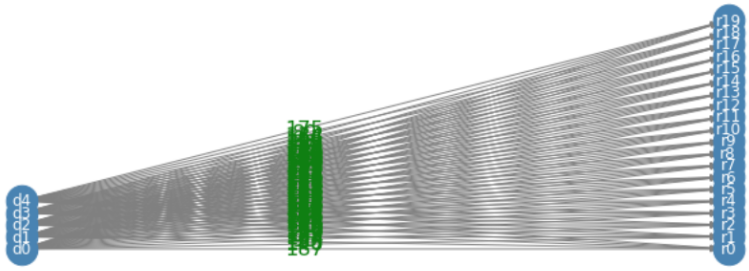
is connected to a rider with negative edge weight, then this means that the driver did not match with anyone.

As we will see from the examples below, the driver benefits more when there is an even match. Otherwise, the intuition behind supply and demand takes place.

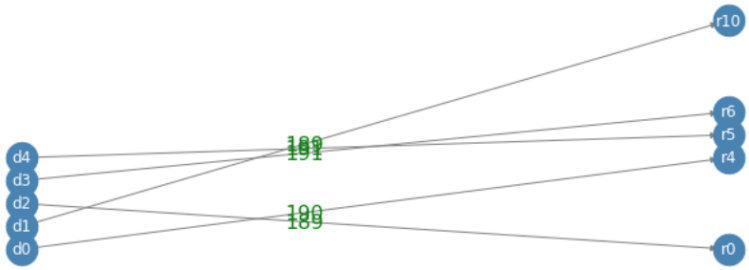
If there are significantly more riders than drivers, then the drivers do not give up as much utility and can usually pair with their preferred choice (highest potential rider).

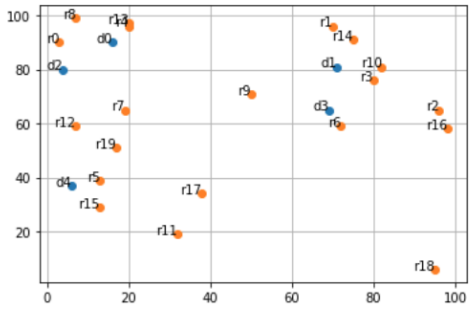
If there are significantly more drivers than riders, then the drivers end up competing against one another for a rider. Therefore, they yield their utility, which means that riders pay less.

```
results with 20 riders and 5 drivers
[('d0', 'r0', {'weight': 187}), ('d0', 'r1', {'weight': 140}), ('d0', 'r2', {'weight
```

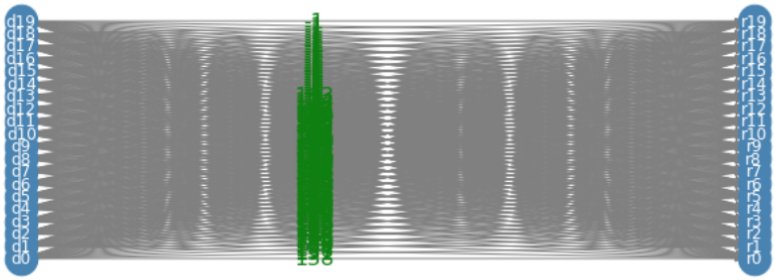


```
Driver discounts [d0,d1,...dn]:  [0, 0, 0, 0, 0]
Drivers recieve [d0,d1,...dn]:  [190, 189, 189, 191, 191]
```

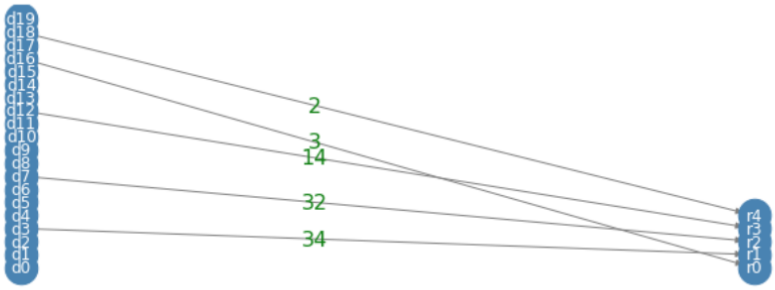


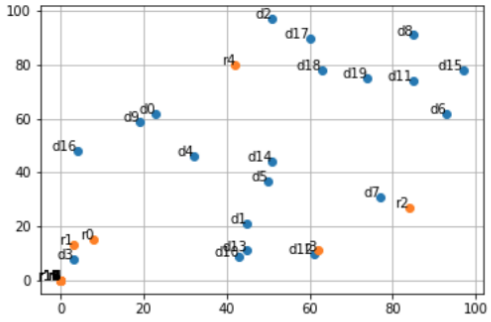


```
results with 5 riders and 20 drivers
[('d0', 'r0', {'weight': 138}), ('d0', 'r1', {'weight': 131}), ('d0', 'r2', {'wei
```

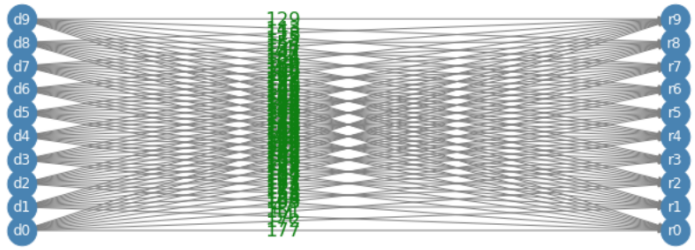


```
Driver discounts [d0,d1,...dn]:  [161, 157, 184, 160, 175]
Drivers recieve [d0,d1,...dn]:  [34, 32, 14, 3, 2]
```

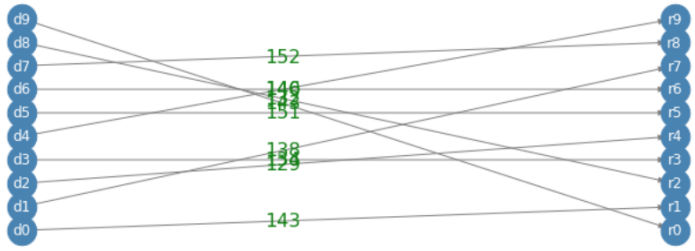


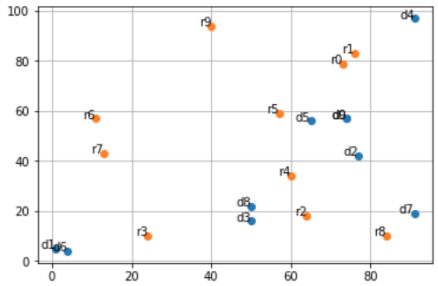


```
[ ] results with 10 riders and 10 drivers
    [('d0', 'r0', {'weight': 177}), ('d0', 'r1', {'weight': 172}), ('d0', 'r2', {'weight':
```




```
Driver discounts [d0,d1,...dn]:  [29, 12, 46, 34, 0, 38, 0, 32, 50, 34]
Drivers recieve [d0,d1,...dn]:  [143, 138, 129, 134, 146, 151, 140, 152, 132, 143]
```





12. There are a few scenarios where drivers will have a preference for where they want to drive. One example would be when a driver heads back home and wants to pick a passenger who is going in the direction of their home. In this case, we could change the cost function to the distance from the rider drop-off to the driver's home. The further away the drop-off is to the driver's home, the bigger the cost. The closer the drop-off to the driver's home, the smaller the cost. Basically the driver is 'subsidizing' a bit of the cost of the ride when the rider lives close to the driver's home.

    Another scenario is when a driver travels to high-density areas (e.g. airports). This means that they are more likely to get a higher valuation ride. To account for this in our algorithm, we could specify high-density quadrants in our map, such that if riders are located in that quadrant, then their valuation function is multiplied by some multiplier.

    □

* **Bonus question 3** Suppose that the city implements a public transportation system. The cost to take the public transportation system is $a+b\cdot dist(\text{initial location}, \text{destintation})$, where a is a fixed base fare and b is some constant factor multiplier. Furthermore, anyone can take this public transportation system from any location.

  (a) Implement a procedure that includes the public transportation option in when computing the stable outcome above. Does a stable outcome always exist in this case?

  (b) Choose different values for a and b and discuss how it affects the prices charged to riders/ profits of drivers.

  (c) (Open ended: variable points awarded based on quality of solution.) Extend your context to include other additional factors that you might find interesting. Summarize what you did and what results you found in your writeup.

Solution:

(a)

(b)

(c)

□

```
import re
import random
import numpy as np
import networkx as nx
#import maxflow

from networkx.algorithms import bipartite
from random import randrange
from scipy.spatial.distance import cityblock

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

Part 4: Implementing Matching Market Pricing. The goal of this
an algorithm for finding market-clearing and VCG prices in a bi
Include your code in matchingmarket.py.

7. Recall the procedure constructed in Theorem 8.8 of the notes to find matching market.

ford_fulkerson max flow code

↳ *1 cell hidden*

(a)

The first step of this procedure involves either finding a perfect matching or a constricted set. Recall t
Now, using your maximum-flow implementation from assignment 2, implement an algorithm that find
constricted set S in a bipartite graph.

```
#perfect matching or constricted?
def match(graph, vcg=None):

  max_graph = nx.Graph.copy(graph)

  #add flow and capacity if not there
  if not nx.get_edge_attributes(max_graph,'flow'):
    nx.set_edge_attributes(max_graph, 0, 'flow')
    nx.set_edge_attributes(max_graph, 1, 'capacity')
```

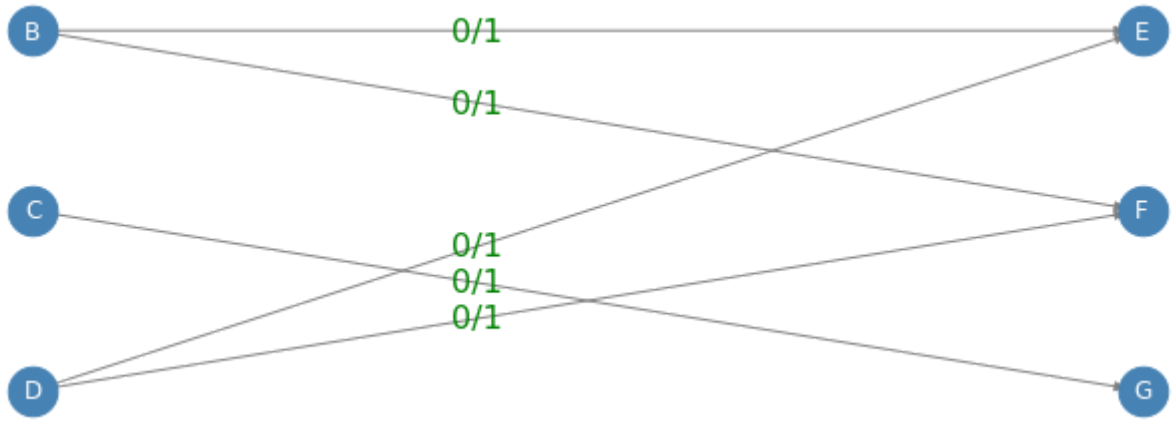
```
#add source node and sink
max_graph.add_nodes_from('ST')

buyers = []
sellers = []

#find buyers and sellers
for edge in max_graph.edges:
  if edge[0] not in buyers:
    buyers.append(edge[0])
  if edge[1] not in sellers:
    sellers.append(edge[1])

#add source edge to buyers and sink to sellers
for node in buyers:
  max_graph.add_edge('S',node, capacity=1, flow=0)

  for node in sellers:
    max_graph.add_edge(node, 'T', capacity=1, flow=0)

flow = ford_fulkerson(max_graph,"S", "T")
#flow, Gnew = max_flow.fordFulkerson(max_graph, S, T)

##check if we have a matching market or not
if (flow == len(buyers) == len(sellers)):
  result = 'matching'

#need to return just constricted set if vcg
elif vcg:
  result = []
  for node in max_graph['S']:
    if not max_graph['S'][node]['flow']:
      for x in max_graph[node]:
        #x is the item being contricted so lets return its buyers
        for con_edge in max_graph.in_edges(x):
          if con_edge not in result:
            result.append(con_edge)

else:
  #find the constricted item nodes
  result = []
  for node in max_graph['S']:
    if not max_graph['S'][node]['flow']:
      for x in max_graph[node]:
        if x not in result:
          result.append(x)


#create new graph showing only matching edges
```

```
  match_graph = nx.DiGraph()
  match_graph.add_nodes_from(buyers+sellers)

  for edge in graph.edges:
    if max_graph[edge[0]][edge[1]]['flow'] and 'weight' in max_graph[edge[0]][edge[1]]
      match_graph.add_edge(edge[0],edge[1], weight= int(max_graph[edge[0]][edge[1]]['w

    elif max_graph[edge[0]][edge[1]]['flow']:
      match_graph.add_edge(edge[0],edge[1])

  if(not result):
    result='matching'
  return result, match_graph
```

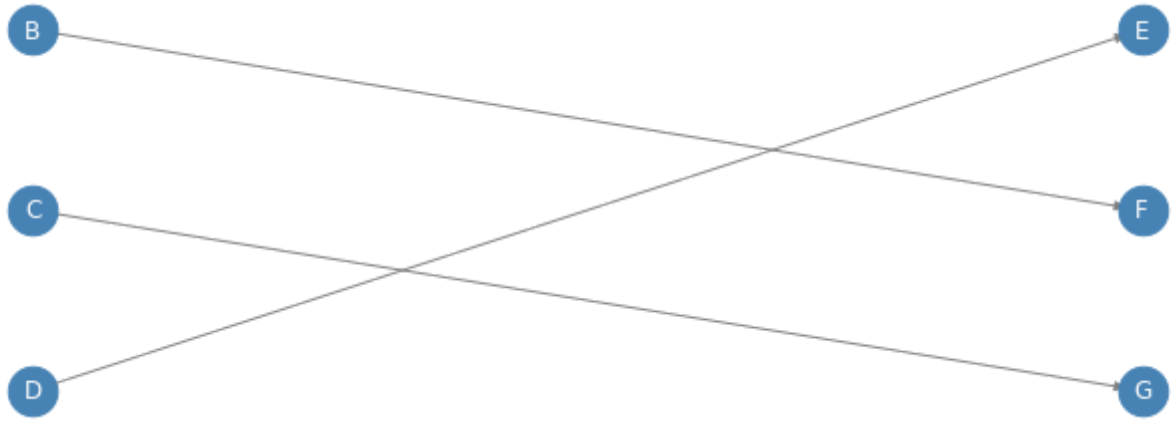
```
graph = nx.DiGraph()
graph.add_nodes_from('BCDEFG')
graph.add_edges_from([
    ('B', 'E', {'capacity': 1, 'flow': 0}),
    ('B', 'F', {'capacity': 1, 'flow': 0}),
    ('C', 'G', {'capacity': 1, 'flow': 0}),
    ('D', 'F', {'capacity': 1, 'flow': 0}),
    ('D', 'E', {'capacity': 1, 'flow': 0}),

])

layout = {
   'B': [1, 2], 'C': [1, 1], 'D': [1, 0],'A': [1, 3],
    'E': [2, 2], 'F': [2, 1], 'G': [2, 0], 'T': [3, 1],
}

graph2 = nx.DiGraph()
graph2.add_nodes_from('ABCDEFG')
graph2.add_edges_from([
    ('B', 'G', {'capacity': 1, 'flow': 0}),
    ('C', 'G', {'capacity': 1, 'flow': 0}),
    ('D', 'F', {'capacity': 1, 'flow': 0}),
    ('D', 'E', {'capacity': 1, 'flow': 0}),


])


print ('Example 1')
print("Pairing Graph")
draw_graph(layout, graph)

result, graph = match(graph)
print("Graph is "+result)
draw_graph(layout, graph)
```

```
draw_graph(layout, graph)

print ('\n\n\n\nExample 2 \n\n\n')

print("Pairing Graph")
draw_graph(layout, graph2)

result, graph2 = match(graph2)
print("Graph is constricted" , result)
draw_graph(layout, graph2)
```

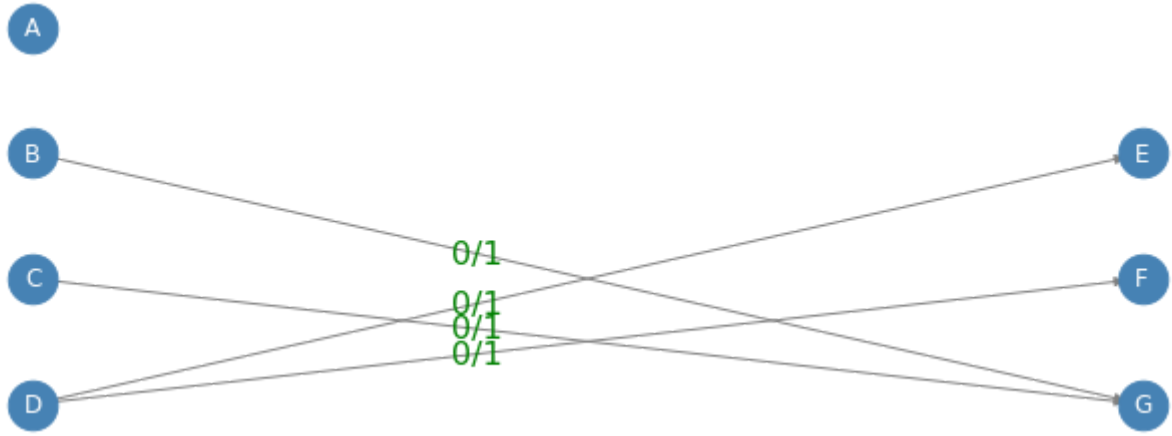
```
Example 1
Pairing Graph
```



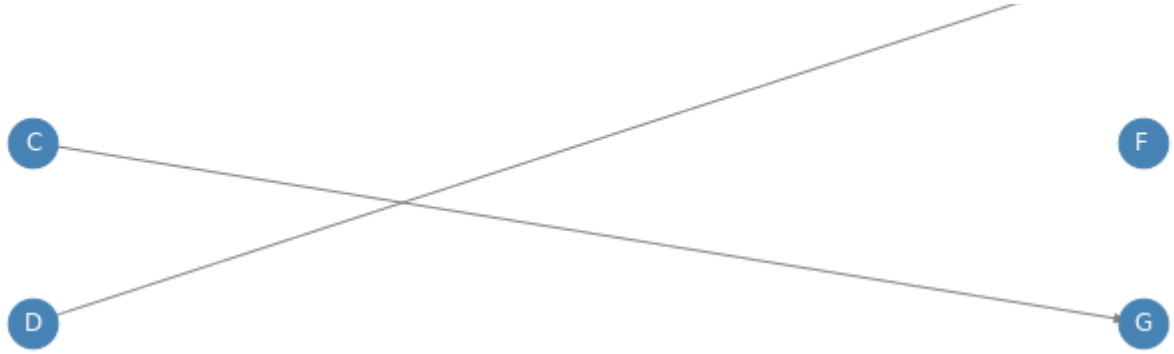

```
Graph is matching
```




```
Example 2


Pairing Graph
```



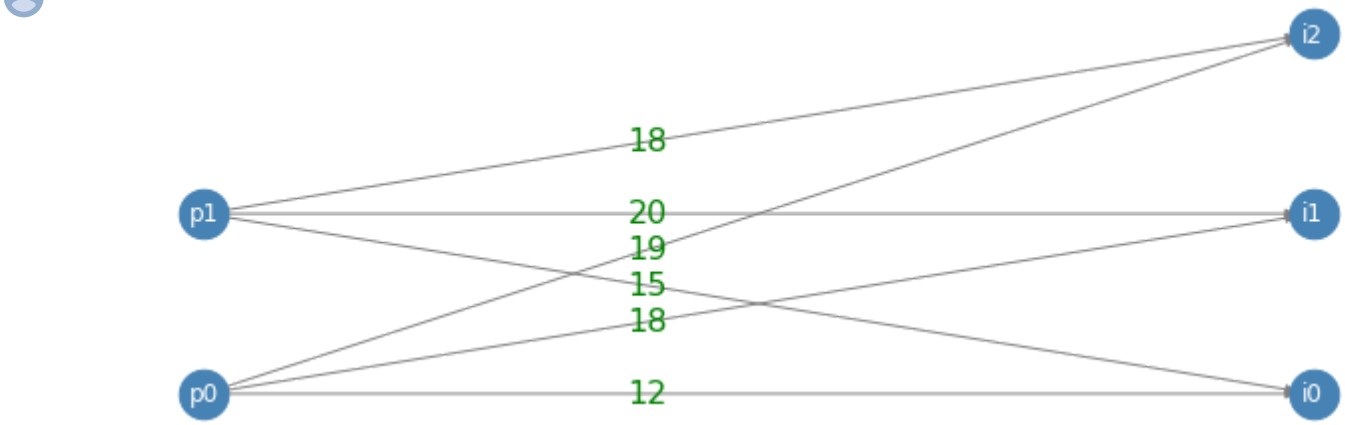

```
Graph is constricted ['G']
```



(b)

Now, given a bipartite matching frame with players, n items, and values of each player for each item, i
market equilibrium.

```
#create a graph with random values from each player to item
 def create_graph(players, items):
  graph = nx.DiGraph()
  edges = []
  for i in range(players):
    for j in range(items):
      edges.append(('p' + str(i),'i' + str(j),random.randint(2,20)))

  graph.add_weighted_edges_from(edges)

  return graph

def get_layout(players,items, uber=None):
  layout = {}

  if (uber):
    for i in range(players):
      layout['d'+str(i)] = [1,i]
    for i in range(items):
      layout['r'+str(i)] = [2,i]
    return layout

  else:
    for i in range(players):
      layout['p'+str(i)] = [1,i]
    for i in range(items):
      layout['i'+str(i)] = [2,i]
    return layout

draw_graph(get_layout(2,3),create_graph(2,3))
```




```
def market_clearing(graph, uber=None):
  #iterativly check if we have an equilibrium then change prices if we dont
  flag = True

  #Cant assume players and items are equal so lets find them
  buyers = []
  sellers = []

    #find buyers and sellers
  for edge in graph.edges:
    if edge[0] not in buyers:
      buyers.append(edge[0])
    if edge[1] not in sellers:
      sellers.append(edge[1])

  #set all item prices to 0
  cost = np.zeros(len(sellers))

  while(flag):

    #create a graph from player to favorite item and run matching market on it
    test_graph = nx.DiGraph()
    test_graph.add_nodes_from(buyers+sellers)

    for player in buyers:
      vals=[]
      for i, item in enumerate(graph[player]):
        vals.append(graph[player][item]['weight']- cost[i])

      #find the favorites (there can be ties)
      max_fav = [x for x, e in enumerate(vals) if e == max(vals)]
      for fav in max_fav:
        test_graph.add_edge(player,sellers[fav], capacity=1, flow=0, weight = graph[pl


    #now check if we are stable
```

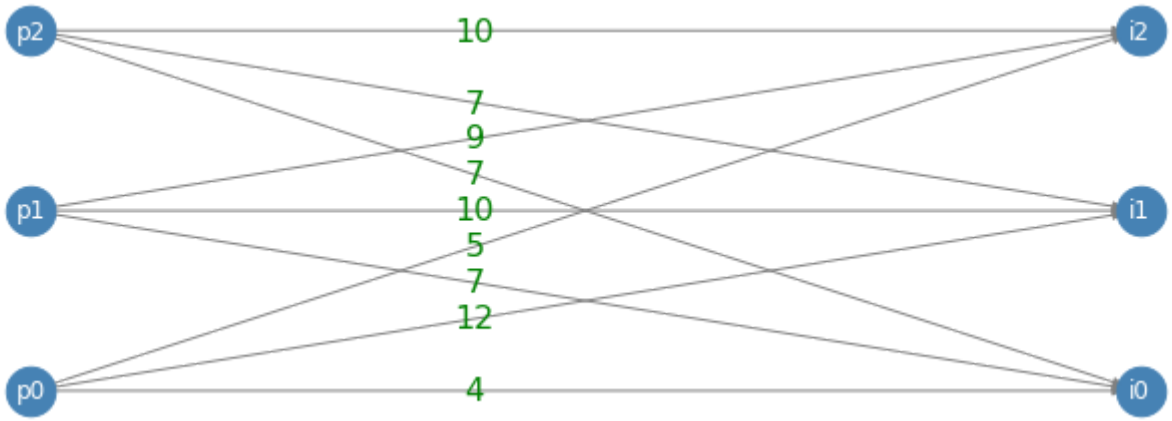
```
    result, matching_graph = match(test_graph)

    if result == 'matching':
      flag = False
      return matching_graph, cost
    else:
      #the result is the constricting items so we need to increase their prices!
      for node in result:

        cost[int(node[1:])] += 1

    #check if all prices are above 0 or not and shift
    if (min(cost)>0):
      print('min cost')
      cost = cost - 1
```

(c)

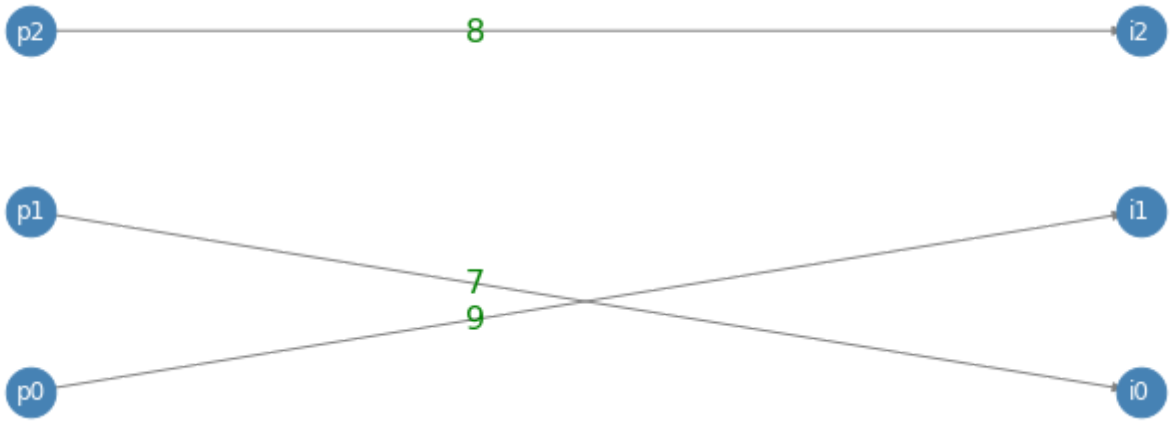
Submit your code along with its output on the matching market frame in figure 8.3 as well as 3 other s choice. Include the input and outputs in a file called p7.txt

```
print('Example input: figure 8.3')


g = nx.DiGraph()
g.add_edges_from([
    ('p0', 'i0', {'weight':4}),
    ('p0', 'i1', {'weight':12}),
    ('p0', 'i2', {'weight':5}),
    ('p1', 'i0', {'weight':7}),
    ('p1', 'i1', {'weight':10}),
    ('p1', 'i2', {'weight':9}),
    ('p2', 'i0', {'weight':7}),
    ('p2', 'i1', {'weight':7}),
    ('p2', 'i2', {'weight':10}),
])
print(g.edges(data=True))
size = 3
draw_graph(get_layout(size,size),g)
g,prices = market_clearing(g)
print('Example output equilibrium')
print (g.edges(data=True),"\n prices (i0,i1,...in): ",prices,'\n')
draw_graph(get_layout(size,size),g)
```

```
Example input: figure 8.3
[('p0', 'i0', {'weight': 4}), ('p0', 'i1', {'weight': 12}), ('p0', 'i2', {'weight
```

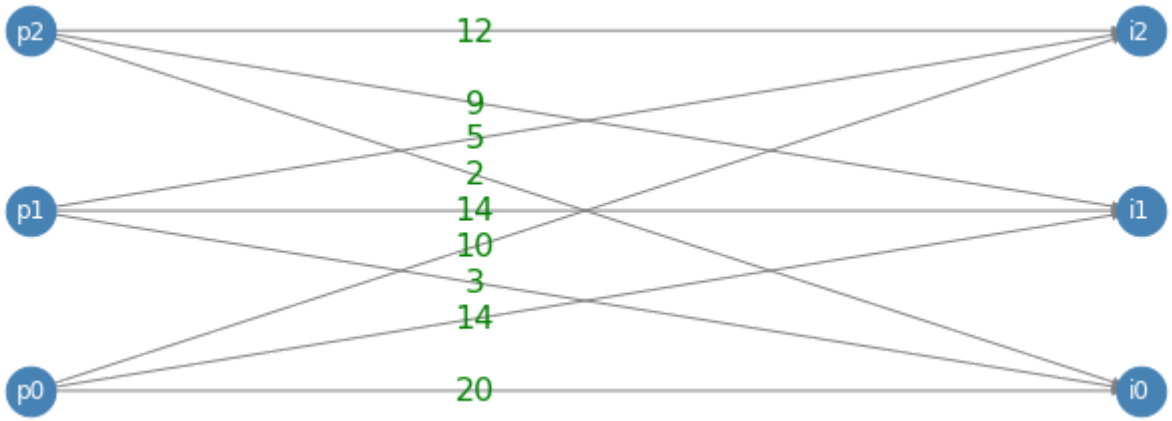



```
Example output equilibrium
[('p0', 'i1', {'weight': 9}), ('p1', 'i0', {'weight': 7}), ('p2', 'i2', {'weight'
 prices (i0,i1,...in):  [0. 3. 2.]
```

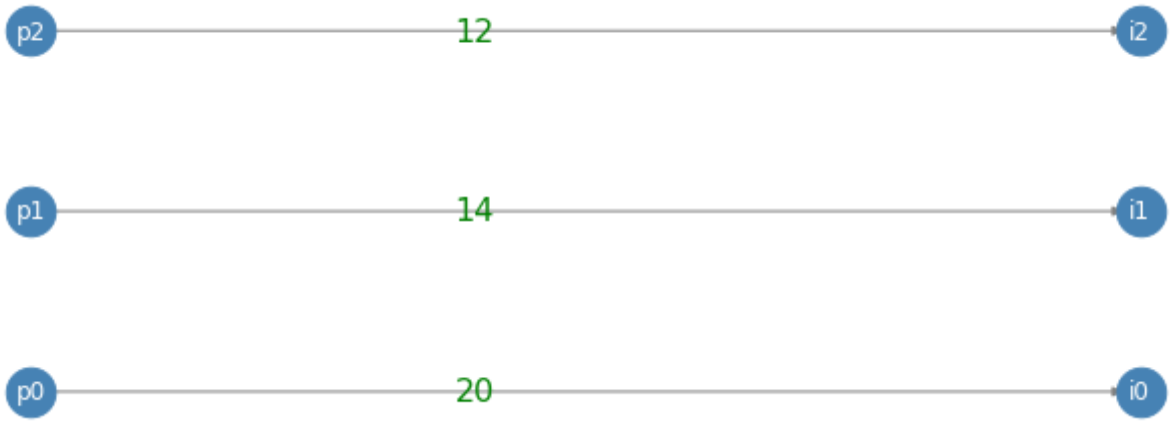



```
#three examples
print('Example input')
size = 3
g = create_graph(size,size)
print(g.edges(data=True))
draw_graph(get_layout(size,size),g)
g,prices = market_clearing(g)
print('Example output equilibrium')
print (g.edges(data=True),"\n prices (i0,i1,...in): ",prices,'\n')

best_lst = [[a, b, data['weight']] for a, b, data in g.edges('p0', data=True)]
print(best_lst[0][1])

#print("DATA", g.edges('p0', data=True)[0])
#graph_copy.edges(node, data=True)
draw_graph(get_layout(size,size),g)
```

```
Example input
[('p0', 'i0', {'weight': 20}), ('p0', 'i1', {'weight': 14}), ('p0', 'i2', {'weigh
```

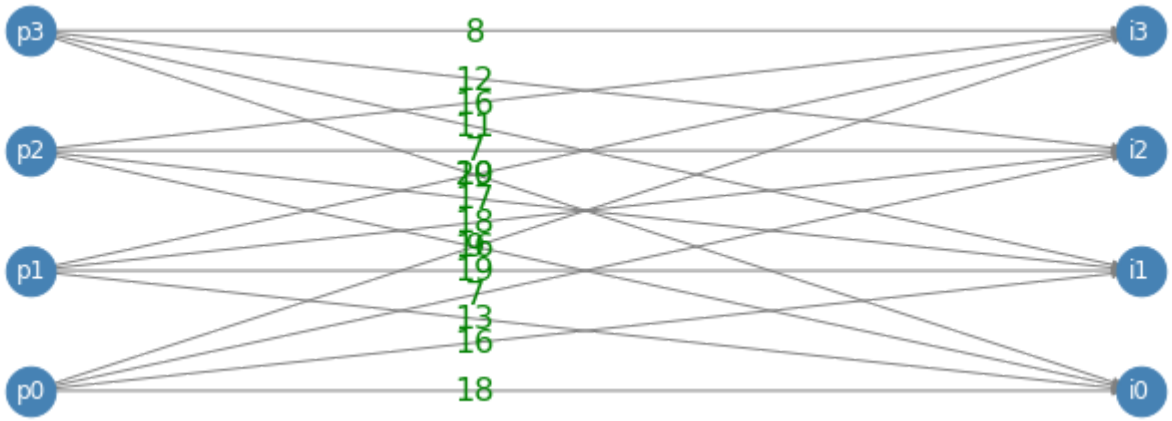



```
Example output equilibrium
[('p0', 'i0', {'weight': 20}), ('p1', 'i1', {'weight': 14}), ('p2', 'i2', {'weigh
 prices (i0,i1,...in):  [0. 0. 0.]

i0
```

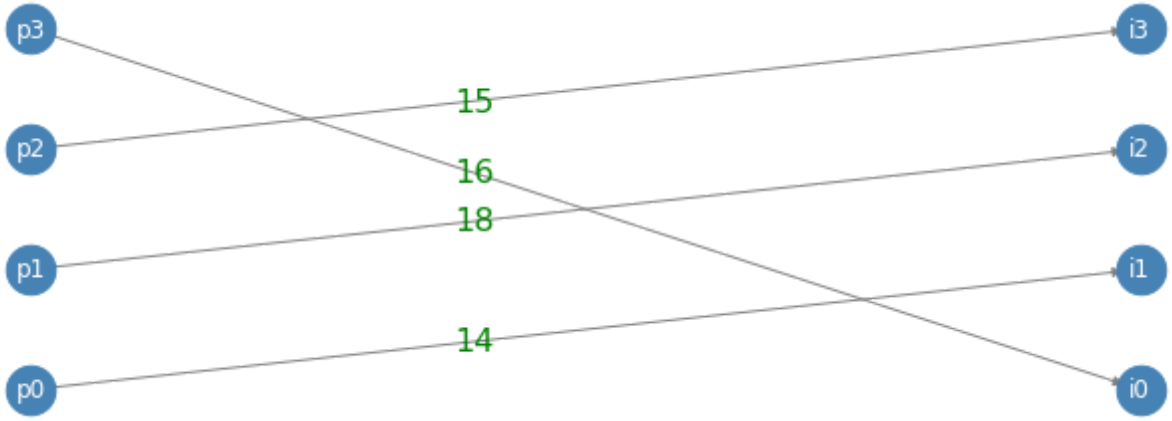



```
print('Example input')
size = 4
g = create_graph(size,size)
print(g.edges(data=True))
draw_graph(get_layout(size,size),g)
g,prices = market_clearing(g)
print('Example output equilibrium')
print (g.edges(data=True),"\n prices (i0,i1,...in): ",prices,'\n')
draw_graph(get_layout(size,size),g)
```

```
Example input
[('p0', 'i0', {'weight': 18}), ('p0', 'i1', {'weight': 16}), ('p0', 'i2', {'weigh
```

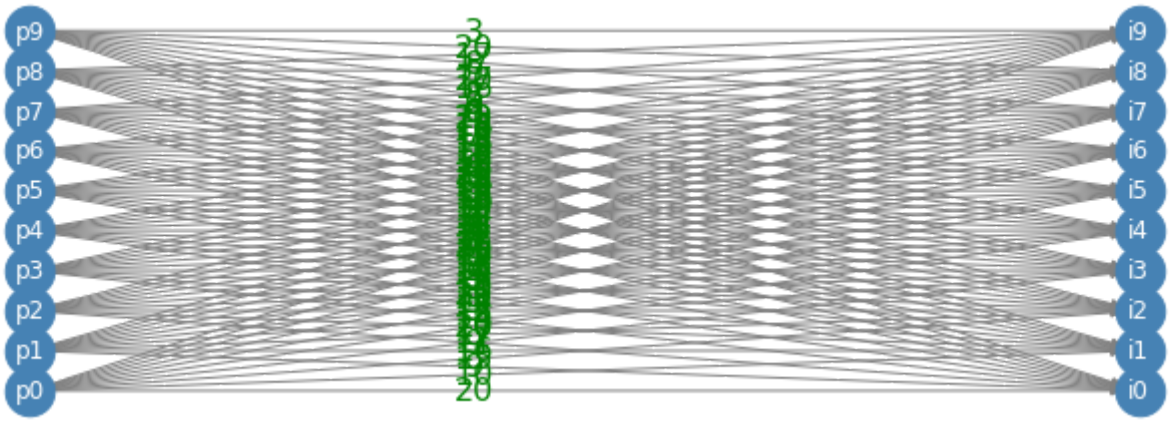



```
Example output equilibrium
[('p0', 'i1', {'weight': 14}), ('p1', 'i2', {'weight': 18}), ('p2', 'i3', {'weigh
 prices (i0,i1,...in):  [4. 2. 0. 1.]
```

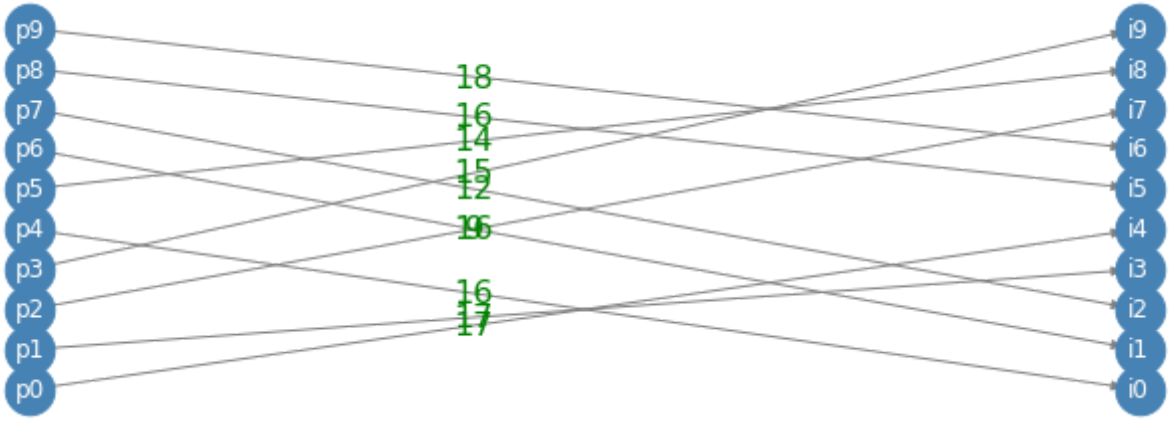



```
print('Example input')
size = 10
g = create_graph(size,size)
print(g.edges(data=True))
draw_graph(get_layout(size,size),g)
g,prices = market_clearing(g)
print('Example output equilibrium')
print (g.edges(data=True),"\n prices (i0,i1,...in): ",prices,'\n')
draw_graph(get_layout(size,size),g)
```

```
Example input
[('p0', 'i0', {'weight': 20}), ('p0', 'i1', {'weight': 10}), ('p0', 'i2', {'weigh
```




```
Example output equilibrium
[('p0', 'i4', {'weight': 17}), ('p1', 'i3', {'weight': 17}), ('p2', 'i7', {'weigh
 prices (i0,i1,...in):  [4. 1. 3. 2. 3. 4. 2. 0. 2. 5.]
```



8. Now, given a matching market frame, we will implement VC(according to the results of Theorem 10.8 in the notes.

(a) In order to do this, we must find the socially optimal outcome. Briefly justify that the outcome that finds is socially optimal (this can be done by simply stating 1-2 theorems from the notes).

Following theorem 8.7(Social optimality of market equilibria), we know that a market equilibrium optir for the proof. .

Theorem 8.7 proof (mostly taken from the book): Since a market equilibrium is a perfect matching in t every buyer must receive an item that maximizes his or her own utility given the prices. Sellers' utilitie p as well. Thus, following corollary 8.6, our equilibrium maximizes social value.

(b) Finally, implement the VCG mechanism and Clarke pivot rule to con produces a positive set of VCG prices in any matching market frame.

```
def social_value(values, matchings, exclude_player, buyers):
  value = 0
  for player in buyers:
    if (player != exclude_player):
      player_matching_data = [[a, b, data['weight']] for a, b, data in matchings.edges
      house_choice = player_matching_data[0][1]
      player_value_data = [[a, b, data['weight']] for a, b, data in values.edges(playe

      for item in player_value_data:
        if (item[1] == house_choice):
          value = value + item[2]

          break

  return value


def social_value_without(values, exclude_player, buyers):
  graph_copy = nx.Graph.copy(values)
  graph_copy.remove_node(exclude_player)

  g_equilibrium, prices = market_clearing(graph_copy)

  value = 0
  for player in buyers:
    if (player != exclude_player):
      player_matching_data = [[a, b, data['weight']] for a, b, data in g_equilibrium.e
      house_choice = player_matching_data[0][1]
      player_value_data = [[a, b, data['weight']] for a, b, data in values.edges(playe

      for item in player_value_data:
        if (item[1] == house_choice):
          value = value + item[2]
          break

  return value


def vcg(input_graph, buyers):
  g_equilibrium, prices = market_clearing(input_graph)

  pricings = list()
  for player in buyers:
    w = social_value(input_graph, g_equilibrium, player, buyers)
    wo = social_value_without(input_graph, player, buyers)
    externality = wo - w
```

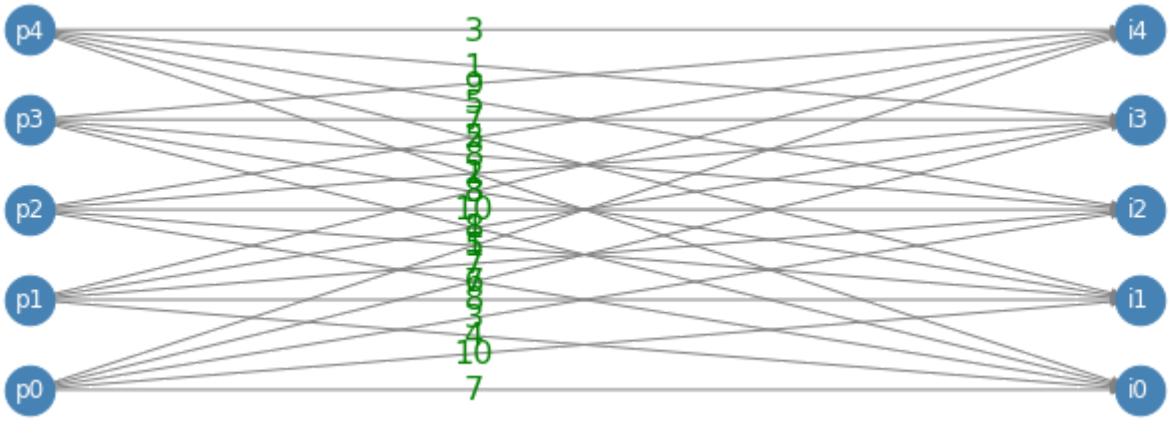
```
    externality = w0 - w
    print("Player {}'s externality: {}".format(player, externality))
    pricings.append(externality)

  return pricings
```

(c) Submit your code along with its output on the matching market frar other small (10-20 node) test examples of your choice. Include the inpu p8.txt

```
graph_102 = nx.DiGraph()

graph_102.add_edges_from([
    ('p0', 'i0', {'weight' : 4}),
    ('p0', 'i1', {'weight' : 12}),
    ('p0', 'i2', {'weight' : 5}),
    ('p1', 'i0', {'weight' : 7}),
    ('p1', 'i1', {'weight' : 10}),
    ('p1', 'i2', {'weight' : 9}),
    ('p2', 'i0', {'weight' : 7}),
    ('p2', 'i1', {'weight' : 7}),
    ('p2', 'i2', {'weight' : 10}),
])

buyers = list()
sellers = list()

#find buyers and sellers
for edge in graph_102.edges:
  if (edge[0] not in buyers):
    buyers.append(edge[0])
  if (edge[1] not in sellers):
    sellers.append(edge[1])

print('example figure 8.3')
print('input:', graph_102.edges(data=True))
res = vcg(graph_102, buyers)
res
```

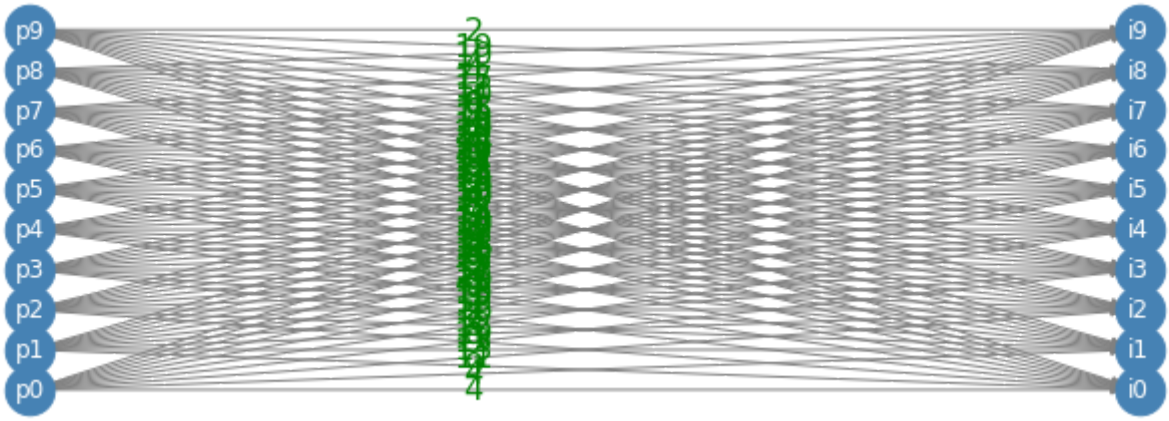
```
example figure 8.3
input: [('p0', 'i0', {'weight': 4}), ('p0', 'i1', {'weight': 12}), ('p0', 'i2', {
Player p0's externality: 3
Player p1's externality: 0
Player p2's externality: 2
[3, 0, 2]
```

```
graph_test_2 = nx.DiGraph()

graph_test_2.add_edges_from([
    ('p0', 'i0', { 'weight' : random.randint(1, 10)}),
    ('p0', 'i1', {'weight' : random.randint(1, 10)}),
    ('p0', 'i2', {'weight' : random.randint(1, 10)}),
    ('p0', 'i3', {'weight' : random.randint(1, 10)}),
    ('p0', 'i4', {'weight' : random.randint(1, 10)}),

    ('p1', 'i0', {'weight' : random.randint(1, 10)}),
    ('p1', 'i1', {'weight' : random.randint(1, 10)}),
    ('p1', 'i2', {'weight' : random.randint(1, 10)}),
    ('p1', 'i3', {'weight' : random.randint(1, 10)}),
    ('p1', 'i4', {'weight' : random.randint(1, 10)}),

    ('p2', 'i0', {'weight' : random.randint(1, 10)}),
    ('p2', 'i1', {'weight' : random.randint(1, 10)}),
    ('p2', 'i2', {'weight' : random.randint(1, 10)}),
    ('p2', 'i3', {'weight' : random.randint(1, 10)}),
    ('p2', 'i4', {'weight' : random.randint(1, 10)}),

    ('p3', 'i0', {'weight' : random.randint(1, 10)}),
    ('p3', 'i1', {'weight' : random.randint(1, 10)}),
    ('p3', 'i2', {'weight' : random.randint(1, 10)}),
    ('p3', 'i3', {'weight' : random.randint(1, 10)}),
    ('p3', 'i4', {'weight' : random.randint(1, 10)}),

    ('p4', 'i0', {'weight' : random.randint(1, 5)}),
    ('p4', 'i1', {'weight' : random.randint(1, 5)}),
    ('p4', 'i2', {'weight' : random.randint(1, 5)}),
    ('p4', 'i3', {'weight' : random.randint(1, 5)}),
    ('p4', 'i4', {'weight' : random.randint(1, 5)}),
])

size=5
draw_graph(get_layout(size,size), graph_test_2)

buyers_two = list()
sellers_two = list()

#find buyers and sellers
for edge in graph_test_2.edges:
  if (edge[0] not in buyers_two):
    buyers_two.append(edge[0])
  if (edge[1] not in sellers_two):
    sellers_two.append(edge[1])

print('input: ',graph_test_2.edges(data=True))
res = vcg(graph_test_2, buyers_two)
res
```



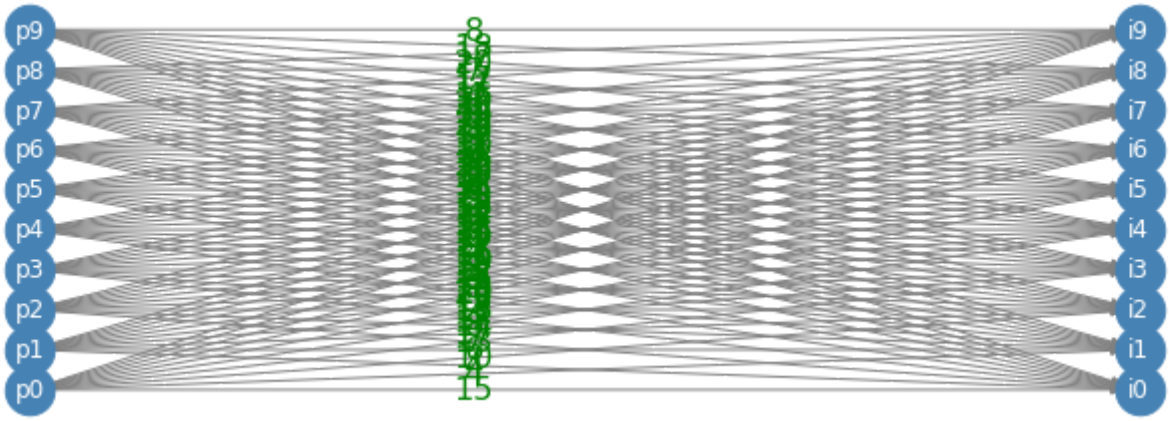

```
input:  [('p0', 'i0', {'weight': 7}), ('p0', 'i1', {'weight': 10}), ('p0', 'i2',
Player p0's externality: 0
Player p1's externality: 2
Player p2's externality: 4
Player p3's externality: 1
Player p4's externality: 4
[0, 2, 4, 1, 4]
```

```
graph_test_3 = create_graph(10,10)
size=10
draw_graph(get_layout(size,size), graph_test_3)

buyers_two = list()
sellers_two = list()

#find buyers and sellers
for edge in graph_test_3.edges:
  if (edge[0] not in buyers_two):
    buyers_two.append(edge[0])
  if (edge[1] not in sellers_two):
    sellers_two.append(edge[1])

print('input: ',graph_test_3.edges(data=True))
res = vcg(graph_test_3, buyers_two)
res
```




```
input:  [('p0', 'i0', {'weight': 4}), ('p0', 'i1', {'weight': 4}), ('p0', 'i2', {
Player p0's externality: 6
Player p1's externality: 5
Player p2's externality: 8
Player p3's externality: 5
Player p4's externality: 6
Player p5's externality: 7
Player p6's externality: 3
Player p7's externality: 0
Player p8's externality: 8
Player p9's externality: 7
[6, 5, 8, 5, 6, 7, 3, 0, 8, 7]
```

```
graph_test_4 = create_graph(10,10)
size=10
draw_graph(get_layout(size,size), graph_test_4)

buyers_two = list()
sellers_two = list()

#find buyers and sellers
for edge in graph_test_4.edges:
  if (edge[0] not in buyers_two):
    buyers_two.append(edge[0])
  if (edge[1] not in sellers_two):
    sellers_two.append(edge[1])

print('input: ',graph_test_4.edges(data=True))
res = vcg(graph_test_4, buyers_two)
res
```




```
input:  [('p0', 'i0', {'weight': 15}), ('p0', 'i1', {'weight': 4}), ('p0', 'i2',
Player p0's externality: 0
Player p1's externality: 2
Player p2's externality: 0
Player p3's externality: 0
Player p4's externality: 1
Player p5's externality: 1
Player p6's externality: 0
Player p7's externality: 0
Player p8's externality: 0
Player p9's externality: 0
[0, 2, 0, 0, 1, 1, 0, 0, 0, 0]
```

9. Now simulate your VCG pricing algorithm in the following cc

(a) First, construct a graph of 20 buyers and 20 items. Assume that “item” i is actually a bundle of ider random value per good (say, from1 to 50; ties can be allowed). You shouldn’t need to do any additiona instead just set each buyer’s value per bundle appropriately. How Should we set these values?

(b) Having set the values accordingly, run your VCG pricing algorithm and turn in the results in a file ca obtained make sense in the context above.

```
buyer_vals = []
bundle_size = []

for x in range(20):
  buyer_vals.append(random.randint(1,50))
  bundle_size.append(random.randint(1,30))

buyer_vals.sort()
bundle_size.sort()

buyer_vals.reverse()
bundle_size.reverse()
```

```
bundle_size=np.arange(20)+1
bundle_size = np.flip(bundle_size,axis=0)
print(buyer_vals)
print(bundle_size)
```

```
[49, 47, 38, 35, 32, 32, 32, 32, 29, 25, 22, 21, 15, 14, 7, 7, 3, 2, 2, 2]
[20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1]
```

```
market = []

for buyer in buyer_vals:
  buyer_bundle_edge = []
  for bundle in bundle_size:
    buyer_bundle_edge.append(buyer * bundle)
  market.append(buyer_bundle_edge)

vcg_graph = nx.DiGraph()
tmp = []
dim =20
for i in range(dim):
  for j in range(dim):
    tmp.append(['p'+str(i), 'i'+str(j), {'weight' : market[i][j]}])
vcg_graph.add_edges_from(tmp)



buyers_two = list()
sellers_two = list()

#find buyers and sellers
for edge in vcg_graph.edges:
  if (edge[0] not in buyers_two):
    buyers_two.append(edge[0])
  if (edge[1] not in sellers_two):
    sellers_two.append(edge[1])
print(market)
print(vcg_graph.edges(data=True))

res = vcg(vcg_graph, buyers_two)
res
```

```
[[980, 931, 882, 833, 784, 735, 686, 637, 588, 539, 490, 441, 392, 343, 294, 245,
[('p0', 'i0', {'weight': 980}), ('p0', 'i1', {'weight': 931}), ('p0', 'i2', {'wei
Player p0's externality: 397
Player p1's externality: 350
Player p2's externality: 312
Player p3's externality: 277
Player p4's externality: 149
Player p5's externality: 181
Player p6's externality: 213
Player p7's externality: 245
Player p8's externality: 120
Player p9's externality: 95
Player p10's externality: 73
Player p11's externality: 52
Player p12's externality: 37
Player p13's externality: 23
Player p14's externality: 9
Player p15's externality: 16
Player p16's externality: 6
Player p17's externality: 0
Player p18's externality: 2
Player p19's externality: 4
[397,
 350,
 312,
 277,
 149,
 181,
 213,
 245,
 120,
 95,
 73,
 52,
 37,
 23,
 9,
 16,
 6,
 0,
 2,
 4]
```

*Bonus Question 2.(Please note that each part of this question than the previous parts; each part will be graded separately, an do the whole thing.)

(a) Implement an algorithm for GSP pricing in a matching-market context. Compare yourVCG prices fr prices when run in the same contexts(with the same randomness). Also, try to find and characterize s are similar, and where they are wildly different. Include a file called bonus-2a.txt displaying input and c

(b) GSP isn't truthful, so interesting things might happen if we run BRD on a GSP matching market auc valuation report). Implement an algorithm that picks a random starting point and runs BRD to attempt happens? Does BRD converge; if so, how quickly? Include a file called bonus-2b.txt displaying input ar

(c)(Very difficult.)Either prove that BRD will converge in a GSP context, or disprove it by counterexamp

Part 5: Exchange Networks for UberWe will construct a simplifi ridesharing app like Uber. Our world will consist of ann×ngrid (t example), and there will be two types of participants, riders an

•A rider R is specified by a current location (x0, y0)∈[n]×[n], a desired destination (x1, y1)∈[n]×[n], and a

•A driverDis specified by a current location (x0, y0)∈[n]×[n].

We define the cost of a matching between a riderRand a driverD,c(R, D), to be the distance from the dr destination (measured via manhattan distance, so the distance from (0,0) to (5,2) is 7).

10. Encode the above example as an exchange network and implemen a graphG= (V, E) where the vertices are the riders and drivers and there rider and driver. What value should we associate with each edge in the

The edge should be the value - cost. Because that is the 'potential utility' of a partnership between ride

Following chapter 8.1: Given a set of players (buyers) X = [n] = {1, . . . , n} and a set of n items Y :

- we associate with each player (buyer) i ∈ X, a valuation function vi :Y → N. For each y ∈ Y , vi(y) c
- we associate with each item y ∈ Y , a price p(y) ∈ N.
- a buyer i who receives an item y gets utility vi(y) − p(y).

Double-click (or enter) to edit

```
# this function makes the 100 by 100 grid and adds drivers/riders with a val
def getGraph(dim,riders,drivers, val):
  r = []
  d = []
  edges = []
```

```
  edges = []

  #populate riders each element is [location,destination,value]
  for i in range(riders):
    #if val is an array we will use the values based on the array values otherwise let

    if (type(val) == int):
      v = val
    r.append([[randrange(dim),randrange(dim)],[randrange(dim),randrange(dim)],v])


  #populate drivers each element is [destination]
  for i in range(drivers):
    d.append([randrange(dim),randrange(dim)])



  #now make edge list with edge value (value-cost) each element is [driver node, rider
  for i in range(drivers):
    for j in range(riders):

      edge_val = r[j][2] - cityblock(d[i],r[j][0])
      edges.append([i,j,edge_val])

  ##add dummy variable add riders if more drivers
  if (drivers>riders):
    #add riders
    for i in range(drivers-riders):
      r.append([[0,0],[100,100],-9999999])
      for j in range(drivers):
        edges.append([j,i+riders,-1])




  return np.asarray(r),np.asarray(d), np.asarray(edges)
```

11. Using your algorithm for matching markets above, implement a pro computes a stable outcome in this context. Note that there may not be and drivers.

```
#use this to show the grid
def showGrid(drivers,riders):
  plt.grid()
  plt.plot(drivers[:,0], drivers[:,1], 'o')
```

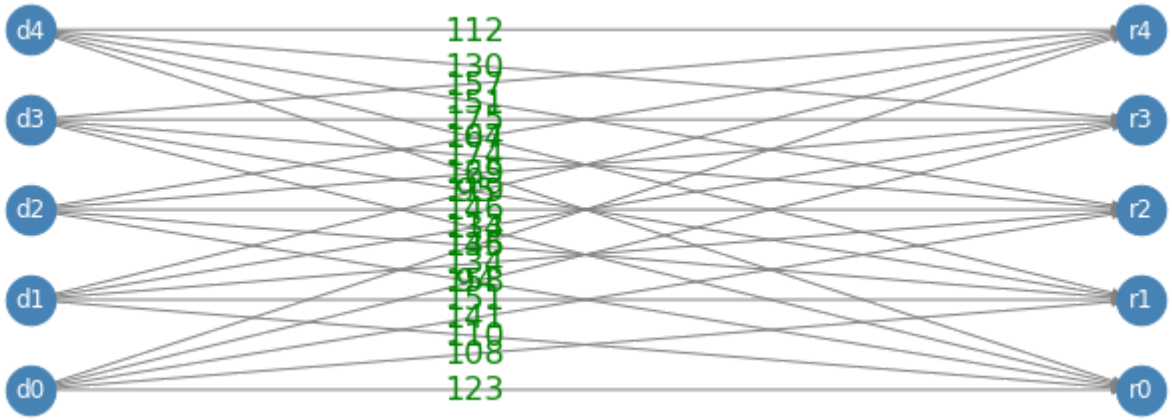
```
    plt.plot([i[0] for i in riders], [i[1] for i in riders], 'o')

    for i,d in enumerate(drivers):
      plt.text(d[0],d[1],'d'+str(i),horizontalalignment='right')

    for i,r in enumerate(riders):
      plt.text(r[0],r[1],'r'+str(i),horizontalalignment='right')

    plt.show()



  #lets make a bipartite graph now for uber
  def make_uber_graph(edges):
    graph = nx.DiGraph()
    for edge in edges:
      graph.add_edge('d'+ str(edge[0]), 'r'+ str(edge[1]), weight = edge[2])

    return graph

  def show_uber(r,d):

    riders, drivers, edges = getGraph(100,r,d,200)

    if (d>r):
      r=d

    uber_graph = make_uber_graph(edges)
    print(uber_graph.edges(data=True))
    draw_graph(get_layout(d,r,True),uber_graph)


    g,not_used = market_clearing(uber_graph)

    #remove dummy riders and their prices

    remove=[]
    prices=[]
    discounts = []
    for edge in g.edges(data=True):
      #if negative remove the dummy
      if edge[2]['weight'] < 0:

        remove.append(edge[1])
      #prices werent working so lets update here
      else:
        #not using this yet but might need it
        new_weight = edge[2]['weight']
        old_weight = uber_graph[edge[0]][edge[1]]['weight']
        prices.append(new_weight)
        discounts.append(old_weight - new_weight)
```

```
  for node in remove:
    g.remove_node(node)


  print("Driver discounts [d0,d1,...dn]: ",discounts)
  print("Drivers recieve [d0,d1,...dn]: ",prices)
  draw_graph(get_layout(d,r, True), g)
  showGrid(drivers, riders[:,0])
```

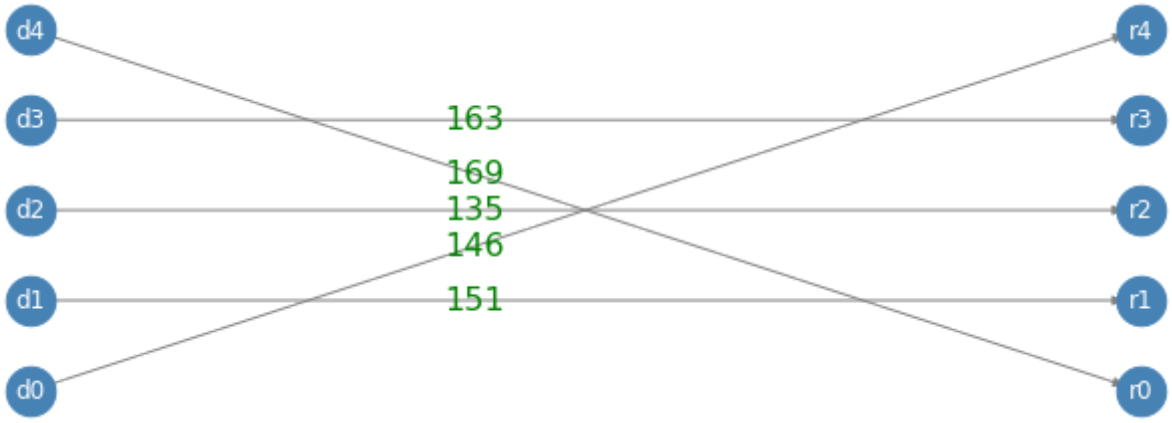
(a) Construct at least two test examples with at least 5 riders and drivers. Discuss your chosen examples. Explain what your results say about how much the riders a drivers are profiting.

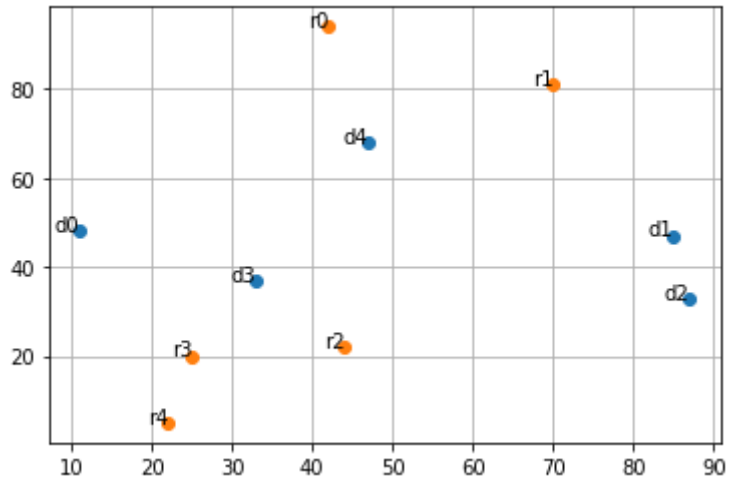
```
show_uber(r=5,d=5)
```

```
[('d0', 'r0', {'weight': 123}), ('d0', 'r1', {'weight': 108}), ('d0', 'r2', {'wei
```



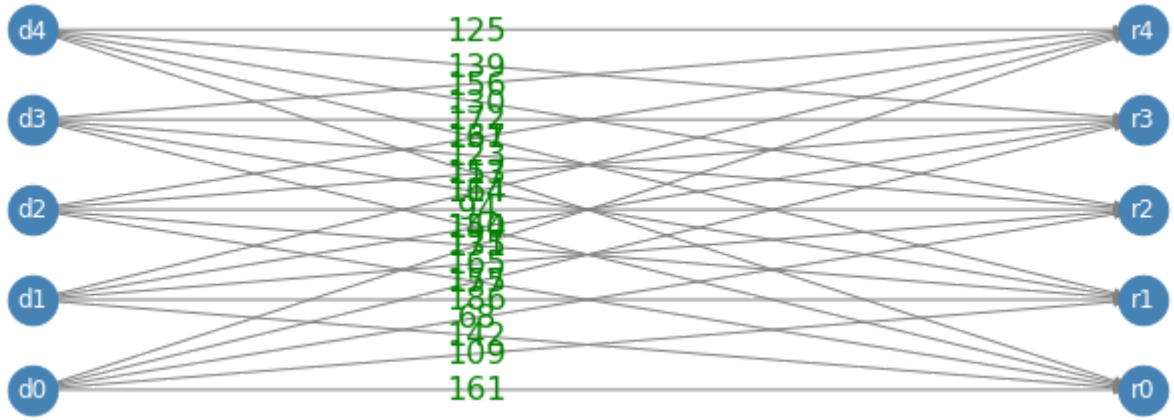

```
Driver discounts [d0,d1,...dn]:  [0, 0, 11, 12, 0]
Drivers recieve [d0,d1,...dn]:  [146, 151, 135, 163, 169]
```



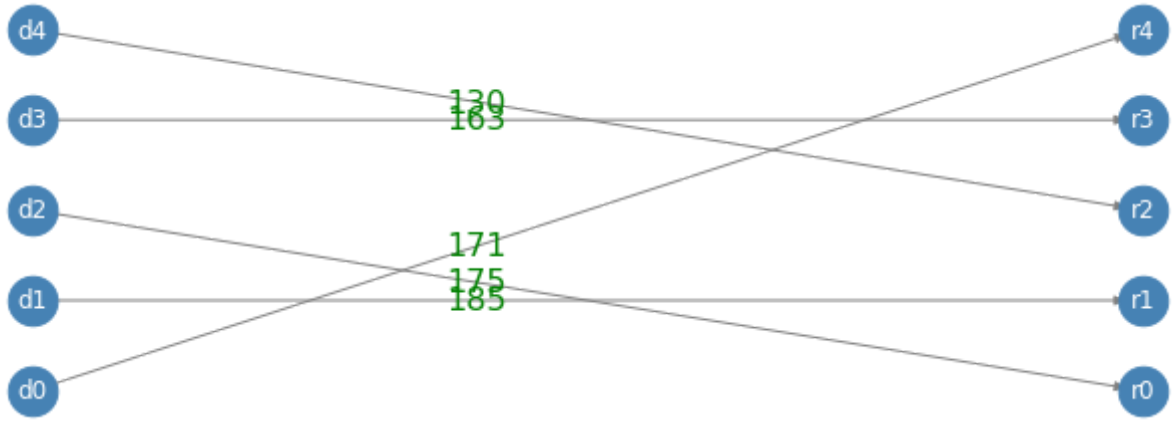


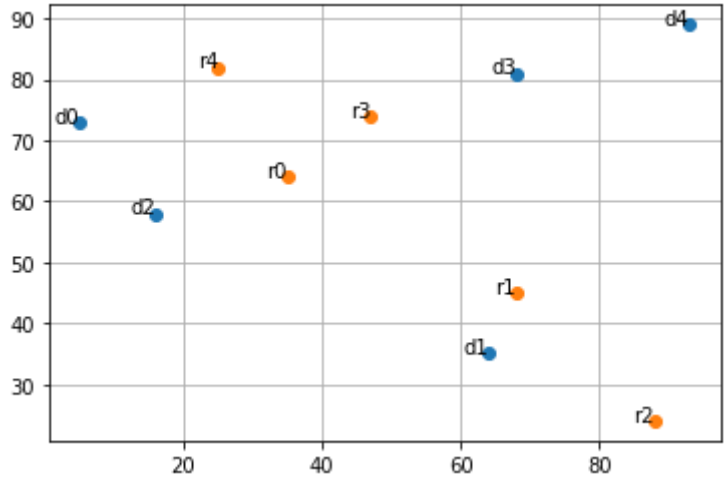

```
show_uber(r=5,d=5)
```

```
[('d0', 'r0', {'weight': 161}), ('d0', 'r1', {'weight': 109}), ('d0', 'r2', {'wei
```




```
Driver discounts [d0,d1,...dn]:  [0, 1, 0, 9, 0]
Drivers recieve [d0,d1,...dn]:  [171, 185, 175, 163, 130]
```



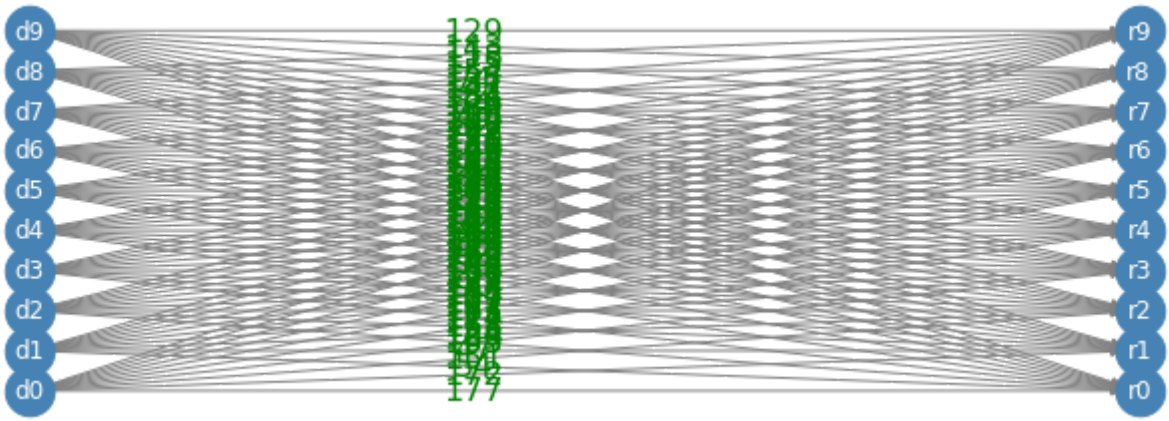


(b) Implement a procedure that on a 100×100 grid (1) generates riders (located ra
value of 100,ddrivers (also located randomly in the grid) and(2) computes a stabl
your procedure many times when r=d (say, 10 each), r is much less than d (say, 5

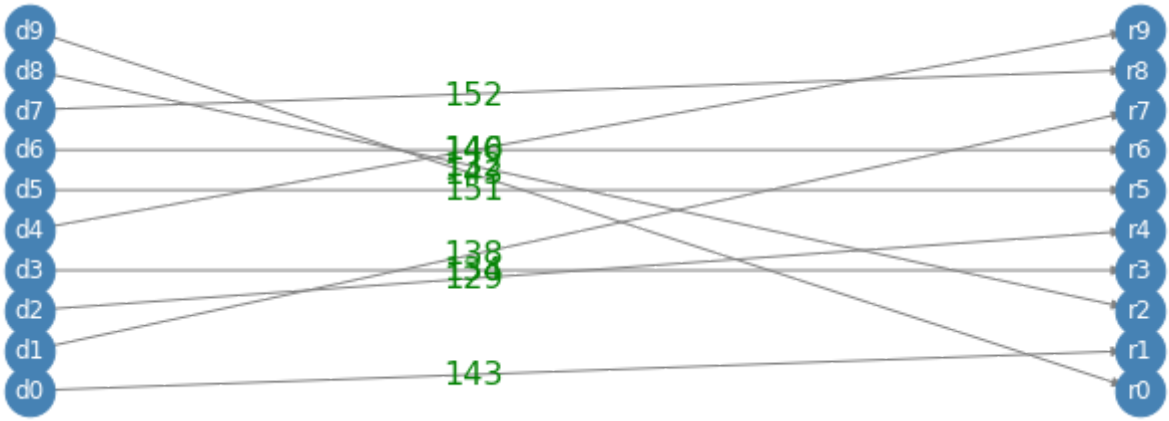
greater than d (say, 20 vs. 5). Discuss your results in detail. Specifically, discuss p ranges of r and d.

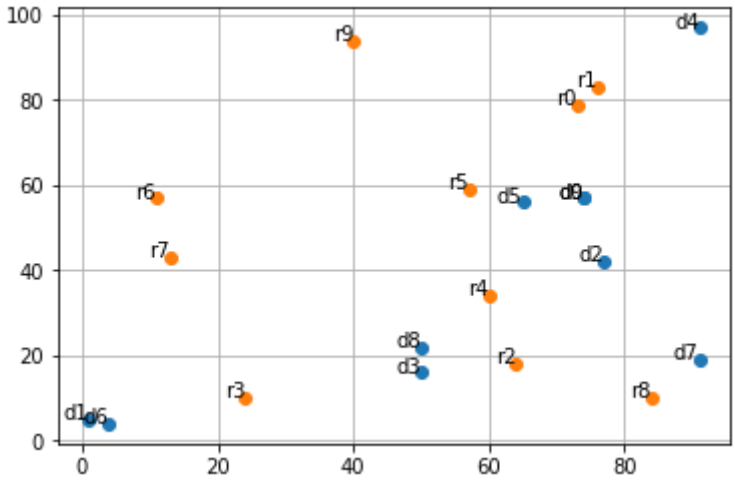
```
##we changed the value to 200 so that we could have our dummy values be a negative nur
print ("results with 10 riders and 10 drivers")
show_uber(r=10,d=10)
```

```
results with 10 riders and 10 drivers
[('d0', 'r0', {'weight': 177}), ('d0', 'r1', {'weight': 172}), ('d0', 'r2', {'wei
```



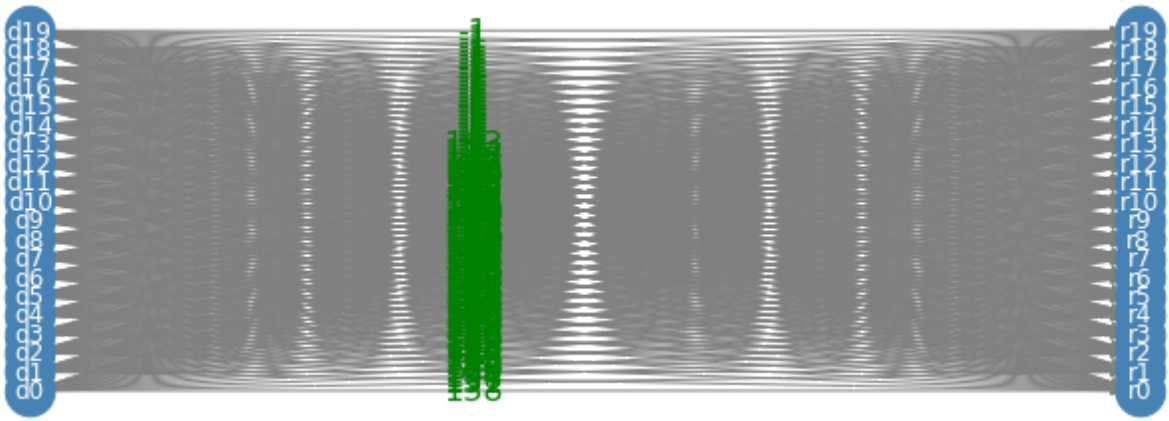

```
Driver discounts [d0,d1,...dn]:  [29, 12, 46, 34, 0, 38, 0, 32, 50, 34]
Drivers recieve [d0,d1,...dn]:  [143, 138, 129, 134, 146, 151, 140, 152, 132, 143
```



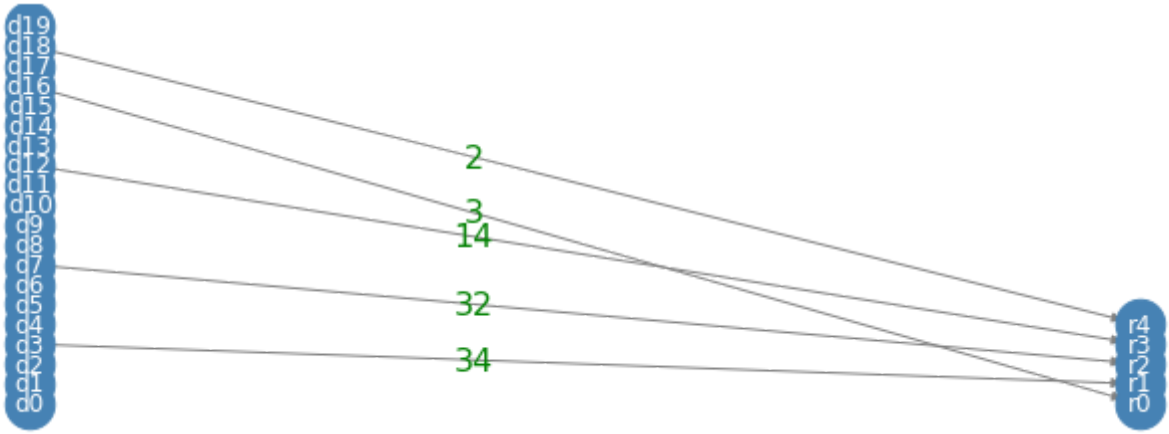


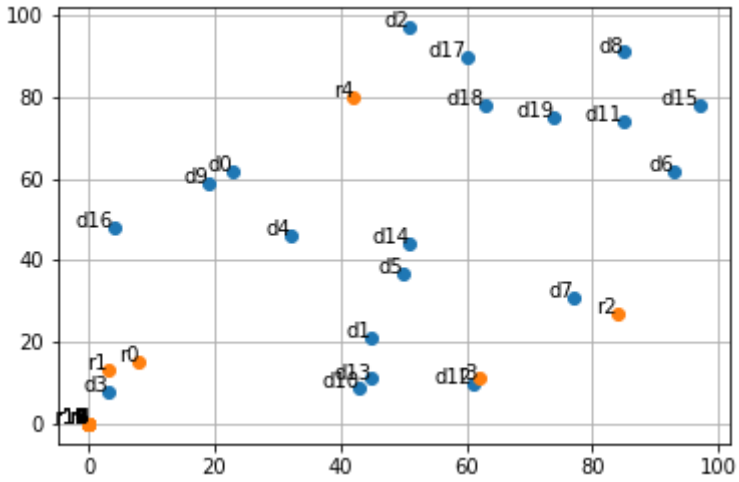

```
print ("results with 5 riders and 20 drivers")
show_uber(r=5,d=20)
```

```
results with 5 riders and 20 drivers
[('d0', 'r0', {'weight': 138}), ('d0', 'r1', {'weight': 131}), ('d0', 'r2', {'wei
```



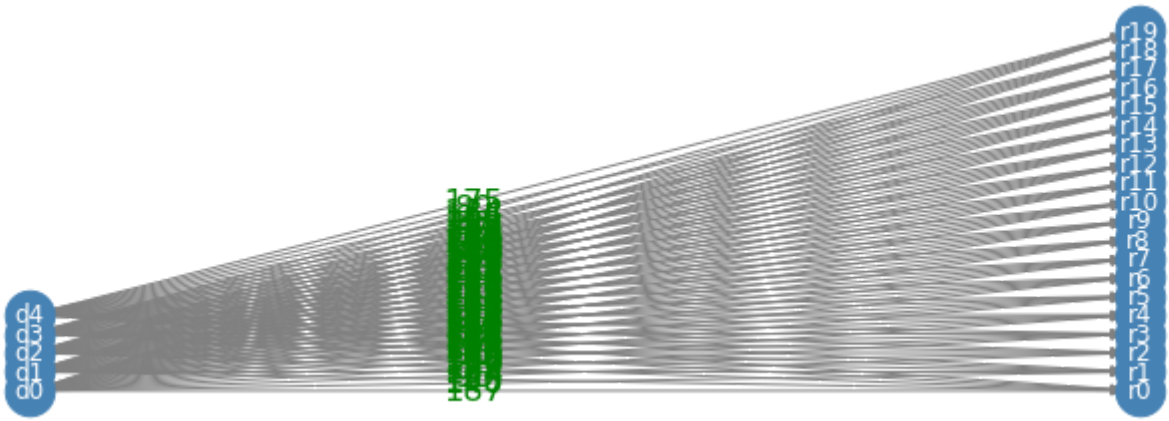
```
Driver discounts [d0,d1,...dn]:  [161, 157, 184, 160, 175]
Drivers recieve [d0,d1,...dn]:  [34, 32, 14, 3, 2]
```



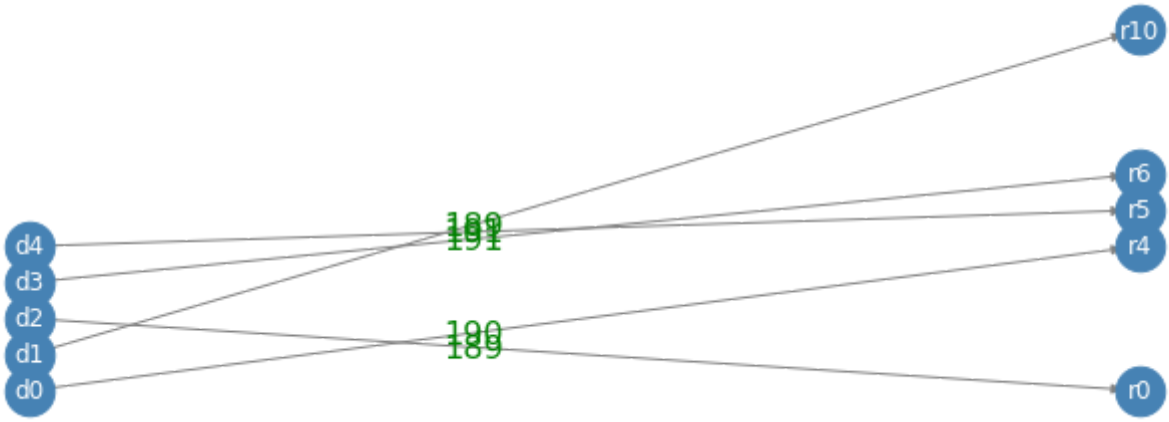


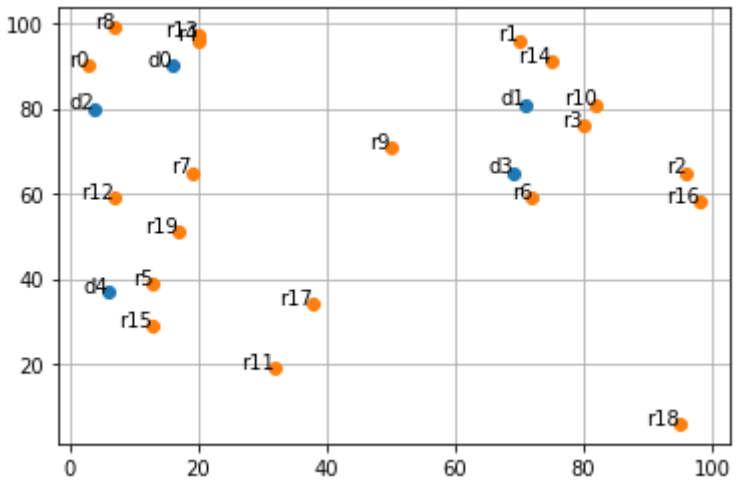
```
print ("results with 20 riders and 5 drivers")
show_uber(r=20,d=5)
```

```
results with 20 riders and 5 drivers
[('d0', 'r0', {'weight': 187}), ('d0', 'r1', {'weight': 140}), ('d0', 'r2', {'wei
```




```
Driver discounts [d0,d1,...dn]:  [0, 0, 0, 0, 0]
Drivers recieve [d0,d1,...dn]:  [190, 189, 189, 191, 191]
```





12. With ridesharing apps, drivers often have preferences for where the
they know they are more likely to get a high value ride from the airport.
how you you could modify the existing setup to include driver's prefere

↳ *1 cell hidden*