ECONOMIC APPLICATIONS OF THE HUNGARIAN ALGORITHM

The purpose of this reading is to explain how the Hungarian algorithm works and how it can be used to do a bunch of things in basic economics. It can find competitive prices, describe negative externalities, characterize equiilibrium in bidding games, even compute payments in Vickrey mechanisms.

The literature on the Hungarian algorithm is large. This paper does two things differently. First, the logic of the algorithm is described using matrix methods, instead of the graph theory based logic that is usually used. Matrix arguments may be more familiar to economists.

Secondly, the paper is primarily focussed on computing and making sense of the weights that are produced as the Hungarian cycles through its methods. These weights aren't normally discussed. Yet they contain all the economic intution that the algorithm provides. The paper includes a python algorithm written by James Yu that does the weight calculations along with the algorithm.

In the first section we describe how the algorithm works. The problem it is designed to solve is to match a finite set I of buyers with a finite set of sellers J in order to maximize

$$\sum_i v\left(i, \mu\left(i\right)\right)$$

where $v\left(i,j\right)$ is the total surplus created by a match between i and j, and $\mu\left(i\right)$ is the partner that i is matched with. A matching function that accomplishes this objective is acc The function $\mu : I \times J$ is one to one but not necessarily onto. It is allowed that $\mu\left(i\right) = \emptyset$

For the moment, we'll assume that I and J have the same number of elements and that $v\left(i,j\right) > 0$ for all i and j. If $v\left(i,j\right) < 0$ they would never be part of an optimal matching. If the sets I and J have different sizes, we can use the methods described below by adding what are effectively phantom buyers of sellers who generate no surplus in any match. We'll leave that discussion until later.

With these assumptions we can restrict attention to matchings that are both one to one and onto by assuming that whenever a pair (i,j) is matched and $v\left(i,j\right)$ is zero then a trade occurs.

Theory. To find prices, we define a series of real numbers $\lambda\left(i\right)$ for each $i \in I$ and $\lambda\left(j\right)$ for each j in J. Call these numbers weights. A set of weights is called a *potential* if for all pairs $\{i,j\}$, $\lambda\left(i\right) + \lambda\left(j\right) \geq v\left(i,j\right)$.

If λ is a potential and μ is a matching, then

$$\sum_i \left(\lambda\left(i\right) + \lambda\left(\mu\left(i\right)\right)\right) \geq \sum_i v\left(i, \mu\left(i\right)\right). \tag{0.1}$$

Theorem. *If μ^* is a matching and λ^* is a potential such that*

$$\lambda^*\left(i\right) + \lambda^*\left(\mu^*\left(i\right)\right) = v\left(i, \mu^*\left(i\right)\right) \tag{0.2}$$

for each i then μ^* is an optimal matching.

Proof. Since every matching μ is one to one and onto

$$\sum_i (\lambda(i) + \lambda(\mu(i))) = \sum_{i \in N} \lambda(i) + \sum_{j \in M} \lambda(j) \equiv \overline{\lambda}.$$

Then in (0.1)

$$\sum_i (\lambda(i) + \lambda(\mu(i))) \geq \sum_i v(i, \mu(i))$$

implies that

$$\sum_i v(i, \mu^*(i)) = \overline{\lambda} \geq \sum_i v(i, \mu(i))$$

for *every* matching μ. □

Potentials in simple problems. Though they look mysterious, it is often quite easy to find potentials. For example, the following matrix represents a matching problem with two buyers to be matched with two sellers. The buyers are the different rows, the sellers different columns. The entries in each cell represent the surplus generated when the buyer associated with the row is matched with the seller associated with the column.

		a	b
	1	100	20
	2	20	100

The optimal matching is obviously to match buyer 1 with seller a and buyer 2 with seller b. It is easy to see that one set of weights that satisfy (0.1) are $\lambda(1) = 100$, $\lambda(a) = 0$, $\lambda(2) = 100$, $\lambda(b) = 0$. Of course, (0.1) isn't very demanding. Another set is $\lambda(1) = 0, \lambda(a) = 100, \lambda(2) = 0, \lambda(b) = 100$.

What the theorem above says is that if we can find a potential λ and a matching μ such that for every i

$$\lambda(i) + \lambda(\mu(i)) = v(i, \mu(i)) \tag{0.3}$$

for all i then μ is an optimal matching. In the example above both sets of weights satisfy (0.3) so matching buyer 1 with seller a and 2 with seller b must be an optimal matching. Evidently the potentials that identify the optimal matching aren't unique.

To see how to convert this stuff into something usable in economics, notice that if the price of seller a's product is 100 and b's product is also priced at 100, each buyer will maximize his or her payoff by purchasing from their appropriate matching parter

A slightly more complicated problem.

	100	50
	80	26

We can try to use the same reasoning as before by setting each row weight in the potential equal to the maximum entry in the row, and each of the column weights to zero.

	(0)	(0)
(100)	100	50
(80)	80	26

The optimal matching could be identified if we could find the appropriate potential. The potential would have to have the property that the sum of the row and column weights in any cell we thought was a match would exactly equal the revenue from the match. There are two cells where this is true. They are starred in the following table.

	(0)	(0)
(100)	100*	50
(80)	80*	26

Every cell in the match must be in a different row *and* column from every other cell in the match. You can also imagine that if you were staring at a giant table with hundreds of rows and columns you might find it pretty difficult to find the optimal matching.

What the Hungarian algorithm does is to allow you to systematically search for the optimal matching and produce the corresponding potential. What will be of most interest here is the potential since all the economic content comes from it. For example, the potential will give us market clearing prices in a competitive equilibrium.

The logic of the algorithm we'll use to find the matching and the prices is both very simple and intuitive, and very hard to implement correctly by hand. The algorithm starts by choosing an assignment of partners μ_0 and a potential λ_0 which together satisfy the equality (0.3), except that when we do this, μ_0 may not be a matching because some participants are assigned more than 1 partner.

The assignment in the table above is an example. Both row players are assigned to the same column player. Yet the sum of the weights is 180 and so is the sum of the surplus values for each pair.

The only problem is that this assignment (indicated by the asterisks in the cells of the table) is not a matching. The way we'll proceed is to try to resolve the conflict between the two row players by re-assigning one of them in the *least cost* way, that is, re-assigning them so that the sum of the surplus associated with the assignment falls by as little as possible.

In the example above, it is easy to find the optimal matching - you just need to compare the sum of the diagonal elements with the sum of the off diagonal elements. 80+59 is bigger than 100+26, so the optimal matching is $(1,2)\,(2,1)$.[1]

So that you have a sense where Theoremcomes in we can illustrate the way the Hungarian algorithm goes about adjusting potentials. We are trying to resolve a conflict between the two row players since they would each like to match with column 1. If we resolve the conflict by moving row 1 to column 2, our total surplus will go down by 50. If we resolve it by moving row 2 to column 2 total surplus falls by 54. Just another way of explaining why the off diagonal is the right match.

The amount 50 is the lowest cost way for us to resolve the conflict. If row 1 where the only person we needed to match, they would end up with a surplus of 100. By adding row 2 to the match, row 1's surplus falls by 50. Now we are doing Econ 100, everyone should pay for any externality they exert. So we replace the column weight for column 1 with the cost 50, and refer to it as the *price* of a match with column 1.

[1]The notation means that the player who is row 1 matches with the player who is column 2, while row 2 matches with column 1.

For the row weight, lets just reduce it until it represents the payoff that row 1 gets by matching with column 2 - 50. To make use of Theorem we need to reduce the payoff for row 2 by the price 50 of a match with column 1. So we replace the row weight for row 2 with 30 since that is the surplus that row 2 will get after paying the price for column 1.

After we make those adjustments the matrix looks like the following:

	(50)	(0)
(50)	100	50*
(30)	80*	26

Once we have made these adjustments, Theorem applies so we know we have our optimal matching.

Of course, we can't do this brute force thinking if we are matching hundreds of buyers and sellers. Indeed when we left the discussion of competitive equilibrium we still didn't know whether or not we could always find one. By converting the logic above into a systematic set of steps we can write an algorithm that will meticulously proceed to calculate these prices in arbitrarily complex problems. Since this algorithm will always converge, we'll have our existence proof.

More General - the Algorithm. .

Lets start with something called a *payoff matrix* A. This is just the revenue function $v(i,j)$ expressed in matrix form. The entry is cell (i,j) is $v(i,j)$. If the number of buyers and sellers is the same, the matrix A will be square.

We'll augment the matrix A by adding an additional row and column, as we did in the example earlier, to represent our potential. To create a valid potential quickly we can make the weight in each row equal to the highest value in the row, while the weight in each column is set to zero. Here is an example we can follow as we describe the algorithm:

	(0)	(0)	(0)	(0)
(150)	125	125	150	125
(175)	150	135	175	144
(255)	122	148	250	255
(180)	139	140	160	180

This is more complicated than the example we discussed earlier. In particular, it isn't obvious what the optimal matching is.

To remind, we'll start with a potential λ (that is, the collection of weights). As we go through the algorithm and create the matching, we'll be continually modifying the potential λ using a couple of concepts. The first is referred to as an *equality graph, E*. Here we'll use the term to mean a collection of cells (i,j) in the matrix (i,j) having the property that

$$\lambda_i + \lambda_j - v(i,j) = 0.$$

So e is a subset of the set of cells in the matrix.

A cell in the payoff matrix can be thought of as an edge in a directed graph pointing from the row player to the column player.

Since we start by assigning row weights to be the largest element in the row, and column weights to be zero, there will always be $|I|$ elements in E at the beginning.

In our example, there are four cells in the equality graph. They are indicated in the following diagram with asterisks in them.

A *partial matching* is a one to one mapping from I to J which isn't necessarily onto. The two cells colored red provide a partial matching. We'll use it and other partial matchings as we go through the algorithm.

	(0)	(0)	(0)	(0)
(150)	125	125	150*	125
(175)	150	135	175*	144
(255)	122	148	250	255*
(180)	139	140	160	180*

A partial matching μ as a collection of cells in the matrix which lie in the equality graph E none of which have any common row or column. Rows or columns that do not contain an element in a partial matching μ are said to be *free.*

The way we created the potential by using the largest value in each row guarantees that we'll start with a partial match. There happen to be two elements in the partial matching here. Depending on the values we might get a complete matching right off the bat. Mostly, though we'll get a partial matching with at least one element.

Notice that in the partial matching above we'll need to resolve some conflicts if we want to create a better matching. For example, both row 1 and 2 want to match with column 3. We are going to use a generalization of the technique we used in the example we did earlier. We'll try to resolve conflicts in the least expensive way.

To explain how to do that in general, we'll need some additional concepts. The next one is something called an *alternating path:*

Definition. Let e be an ordered collection of cells in E that begins with a cell e_0 in a free row. If there is no cell in the equality graph in a free row, let e_0 be any of the cells (i,j) in a free row i such that $|\lambda_i + \lambda_j - v(i,j)| = \delta$ is smallest and subtract δ from the row weight for row i. Then cell e_0 will be in the new equality graph and we can use it to create a single element path $e = \{e_0\}$.

The collection $e = \{e_0, e_1, e_2 \ldots, e_t\}$ is called an alternating path if

- e_k and e_{k+1} are alternately in the same column or row (which means if e_k and e_{k+1} are linked by the same row, then e_{k+1} and e_{k+2} are linked together within a column);
- $e_k \in \mu$ if and only if k is odd;

An alternating path from e_0 staring from a free row whose terminal cell e_t is in a free column is called an *augmenting path.*

There are a couple of alternating paths in the example above. For example, the cell $(1,3)$ is in E and lies in a free row. The cell $(2,3)$ is in E and is part of a partial matching. So the sequence $(1,3),(2,3)$ is an alternating path. It isn't an augmenting path because none of the elements in row 2 are in E.

The reason an augmenting path is interesting is that you can add a new cell to your matching if you find an augmenting path. The simplest example is a trivial augmenting path that starts in a free row that is also in a free column. In other words you can add the cell directly to the existing matching.

Otherwise, all the odd elements of e will be part of a partial matching μ. If e is an augmenting path, then a new (partial) matching $\dot{\mu'}$ can be created consisting of all the even elements of e. Furthermore $|\mu'| = |\mu| + 1$. We'll refer to this as *closing*

the augmenting path and denote the closed version of an augmenting path e as $\overline{e}$. An augmenting path is non-degenerate if it has 3 elements or more.

Proposition. *If $e = (e_0, \ldots, e_t)$ is an augmenting path constructed from matching μ and equality graph E, then there is a new matching μ' given by $\{e_{2k}\}_{k=0,t/2}$ such that $|\mu'| = |\mu| + 1$. If c_i is a column used by μ then c_i is also used in μ', so that closing e will add a new column to the matching.*

Proof. The cells $\{e_{2k+1}\}_{k=0,\frac{t}{2}}$ form a partial matching so that, in particular, each cell lies in a different column. In an augmenting path that starts in a free row, each pair $\{e_{2k}, e_{2k+1}\}$ lie in the same column for each $k < \frac{t}{2}$. Since the new matching is $\{e_{2k}\}_{k=0,\frac{t}{2}}$ each column used in the initial matching is used in the new one. Since the augmenting path ends in a free column, one new column is used. □

Notice that closing an augmenting path gives a sequence that consists of the original augmenting path with elements of μ' adjusted to so that if e_k is an element of $\overline{e}$ and k is even, then e_k contains a match. The terminal element of $\overline{e}$ sits in the column that is added to the matching when the path is closed.

Now in our example above $(1,3),(2,3)$ is an alternating path, but it isn't an augmenting path. We can turn it into one by using a process called *relabeling*.

We can illustrate the relabelling idea by using a single alternating path.

Let i be a free row, and e an alternating path that uses that row (i.e., contains a cell in the row). We'll also assume that the cell in the alternating path that uses the row lies in a column that already contains a match. Otherwise the alternating path would be an augmenting path so we could just close it.

Define $C(e) = \{j : \exists i; (i,j) \in e\}$. This is just the set of columns that have a cell in the alternating path e. Let $R(e) = \{i : \{\exists j : (i,j) \in e\}\}$ be the set of rows used by the alternating path e.

In our example above, row 1 is free and there is just one alternating path that uses that row - the path $(1,3),(2,3)$. Then $C_i = \{3\}$,while $R_i = \{1,2\}$.

Now find:

$$\delta = \min\{\lambda_i + \lambda_j - v(i,j) : i \in R(e); j \notin C(e)\} \tag{0.4}$$

and create a new potential λ as follows:

$$\lambda_i' = \begin{cases} \lambda_i - \delta & i \in R(e) \\ \lambda_i & \text{otherwise.} \end{cases} \tag{0.5}$$

Similarly for each column j

$$\lambda_j' = \begin{cases} \lambda_j + \delta & j \in C(e) \\ \lambda_j & \text{otherwise.} \end{cases} \tag{0.6}$$

In our example above, (0.4) says that we have to check two rows $\{1\}$ and $\{2\}$. Then we need to scan across those rows to check elements that aren't in column 3. So lets check cell $(1,1)$. Summing the row and column weights gives us 150, while the entry in our cell is 125 - a difference of 25. No matter which of the other cells we check we won't find a difference smaller than 25. So we know that $\delta = 25$. Then (0.5) says that we should subtract 25 from the row weight for rows 1 and 2. The (0.6) says we need to add 25 to the column weight for column 3.

The result looks like this:

	(0)	(0)	(25)	(0)
(125)	125*	125*	150*	125*
(150)	150*	135	175*	144
(255)	122	148	250	255*
(180)	139	140	160	180*

Notice that we added 5 new elements to the equality graph E. We now have three augmenting paths: $(1,1)$ by itself, $(1,2)$ by itself, and the more interesting $(1,3),(2,3),(2,1)$. For each of the singleton cells we'd just add a new match in the corresponding cell to close the path. For the last augmenting path we would close it by moving the match in cell $(2,3)$ to cell $(1,3)$, then create a new match in cell $(1,3)$.

This simple step basically explains relabelling and how you can use it to find a new match. You can choose to close any of the three augmenting paths. Try each of them in turn as you practice. Here I'll close the degenerate augmenting path $(1,1)$ (because it creates the most interesting augmenting path in the next step).

This completes one *iteration* of the algorithm that we'll develop below. The state of the algorithm at this point looks like this:

	(0)	(0)	(25)	(0)
(125)	125*	125*	150*	125*
(150)	150*	135	175*	144
(255)	122	148	250	255*
(180)	139	140	160	180*

To complete the algorithm we need to generalize the relabelling step slightly. We've described how to relabel using a single alternating path e. It is possible that a free row may be used by more than one alternating path. If so, we want to relabel all these paths at the same time. To do this start by formally describng the process of relabeling based on a free row i.

If i is a free row, let $\mathcal{E}_i$ be the set of all alternating paths that use i. Let

$$\mathcal{C}_i = \cup_{e\in\mathcal{E}_i} C(e)$$

and

$$\mathcal{R}_i = \cup_{e\in\mathcal{E}_i} R(e).$$

Then we say we *relabel based on free row* i when we create a new labelling λ' from initial labels λ by applying (0.4), (0.5) and (0.6) by replacing $C(e)$ with $\mathcal{C}_i$ and $R(e)$ with $\mathcal{R}_i$ wherever they occur.

Now we can describe the full algorithm:

(1) using the current μ and λ, close augmenting paths to get a new μ'; stop if μ' is complete, otherwise go on to step 2;
(2) if there is a free row i used by one or more alternating paths then relabel based on row i to get a new E' and λ' then go back to step 1; else go to step 3
(3) if i is a free row, relabel using $\mathcal{R}_i = \{i\}$ and $\mathcal{C}_i(e) = \{\emptyset\}$ then go back to step 1.

In our example, we just completed step 1. So lets continue the iteration by choosing an alternating path starting in a free row. In particular, lets use the only remaining free row 4, and the only alternating path that uses it $(4,4)(3,4)$. Evidently $\mathcal{C}_4 = C((4,4)(3,4)) = \{4\}$ while $\mathcal{R}_4 = R((4,4)(3,4)) = \{3,4\}$. So we have to check

each of the cells that are in row 3 and 4 but aren't in column 4. Check to verify that $\delta = 30$. Then relabel by adding 30 to the column weight for column 4 and subtract 30 from the row weights in row 3 and 4. The result is this:

	(0)	(0)	(25)	(30)
(125)	125*	125*	150*	125*
(150)	150*	135	175*	144
(225)	122	148	250*	255*
(150)	139	140	160	180*

Notice that the cell $(3,4)$ that we used to compute δ has been added to the equality graph E. Notice that the new weight for column 4 is 30.

Now we have a nice augmenting path. It starts in a free row 4 - $(4,4)$ $(3,4)$ $(3,3)$ $(2,3)$ $(2,1)$ $(1,1)$ $(1,2)$ - which ends in a free column. Make sure you understand how to get the rest of the match by closing the augmenting path. Also make sure to check that the sum of the row and column weight is at least as large as the value in the cell for every cell in the matrix.

The optimal matching and all the weights are given in this table:

	(0)	(0)	(25)	(30)
(125)	125*	125*	150*	125
(150)	150*	135	175*	144
(225)	122	148	250*	255*
(150)	139	140	160	180*

A couple of comments. In step 2 of the algorithm, the computations involving (0.4), (0.5) and (0.6) check cells that are in rows in $\mathcal{R}_i$ and columns that *aren't* in $\mathcal{C}_i$. None of the cells in this collection can be in the equality graph. The reason is that if they were, they would lie in a row that is already part of a match. So they would be part of an existing alternating path which should have been included when defining $\mathcal{R}_i$ and $\mathcal{C}_i$.

After relabelling this new cell will be in the equality graph. If it is in a free column it will complete an augmenting path and add a new match to the partial matching. If it is in a column that is already occupied, it will extend and existing alternating path increasing the number of elements in $\mathcal{C}_i$ and forcing the algorithm to find another cell to add during the next relabelling. Since there are only finitely many cells for the algorithm to check, it follows that repeated relabelling will always find a cell in a free column. Equivalently, relabelling must find an empty column and use it to augment any partial matching.

Competitive equilibrium. Recall from our previous reading how a competitive equilibrium is constructed. There are buyers and sellers. Buyers' payoffs from buying from seller j are given by $u_i(j)$.

Let u_j be the utility value that seller j gets if they keep their good.[2] If i and j trade at a price p_j, their payoffs are $u_i(j) - p_j$ and $p_j - u_j$ respectively. A competitive equilibrium is a vector of I prices and a matching $\mu : I \to J$ such that for each $i \in i$

$$u_i(\mu(i)) - p_{\mu(i)} \geq \max(u_i(j) - p_j)$$

and $p_j \geq u_j$, for all i and j.

[2]If seller j is providing a service, then u_j would be their utility for leisure. If seller j is holding an asset, then u_j could be the sales price for that asset on some external market.

To use this with the Hungarian algorithm, we can just start by defining

$$v(i,j) = u_i(j) - u_j.$$

Recall our assumption that $v(i,j) \geq 0$.

Theorem. *Let λ and μ be the matching determined by the Hungarian algorithm as described above when the surplus matrix is given the collection $\{v(i,j)\}_{ij}$. Then competitive equiliibium prices are given by $p_j = u_j + \lambda_j$*

Proof. The argument goes as follows: for each i

$$\begin{gathered}(u_i(\mu(i)) - u_{\mu_j} - \lambda_{\mu(i)}) - (u_i(j) - u_j - \lambda_j) = \\ v(i,\mu(i)) - \lambda_{\mu(i)} - \lambda_i - (v(i,j) - \lambda_j - \lambda_i) = \\ -(v(i,j) - \lambda_j - \lambda_i) \geq 0.\end{gathered}$$

□

The reason is that by Theorem 0.1 the first three terms sum to zero, while the final result is positive because the weights form a potential.

For further interpretation, lets write out the fact that the the optimal match lies in the equality graph:

$$v(i,\mu(i)) - \lambda_{\mu(i)} - \lambda_i = 0.$$

This says that

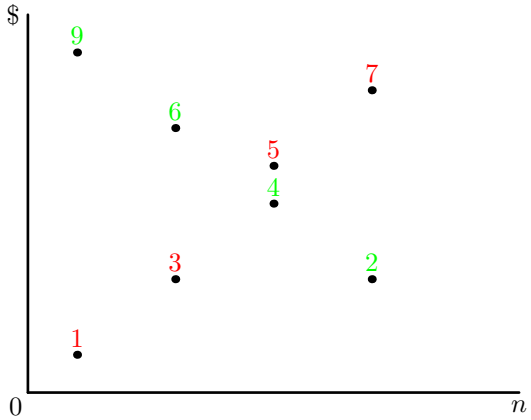
$$v(i,\mu(i)) - \lambda_{\mu(i)} = \lambda_i$$

This says that total surplus less the price the buyer is payed is equal to the row weight for that buyer. The row weights, in other words, represent the share of the total surplus that goes to the corresponding buyer or seller.

Double Auction. In a double auction, every buyer values each seller equally, though, of course, every buyer has a different valuation. Sellers all have different costs. The we can just write $u_i(j) = u_i$ and $v(i,j) = u_i - u_j$

It wouldn't make sense to do this in practice since there are much simpler ways to allocate in a double auction. However it illustrates many of the useful conceptual features of the method.

Here is a simple example of a double auction in diagrammatic form:



The bids of the buyers are given by the green numerals. They have been sorted in descending order. The asks of the sellers are in ascending order and are given by the red numerals.

A seller's offer double auction awards a unit of the good to each of the four highest bids and asks and sets a price equal to the fifth highest bid or ask. So from the picture, it is clear that the buyers with bids 9 and 6 should trade with the sellers whose asks are 1 and 3. Those trades should occur at price 4. This is one of many possible double auction pricing mechanisms.

To set this up as a matching problem using the Hungarian algorithm, we can start by defining the surplus associated with each matched pair as being equal to the difference $v(i,j) = b_i - a_j$, the difference between the bids and asks.

Of course we don't want buyers and sellers to trade when they jointly have negative surplus. We can accommodate this by adding fictional matching partners for each buyer and seller with whom they jointly earn 0 surplus. Matching with a partner with whom there is zero surplus is the same as not trading.

Appendix.

0.0.1. *A python ALgorithm.* Instead of trying to work this out by hand, here is some python code that you can use to work it out for yourself.

```
import numpy as np
z = np.array([9,6,4,2])
y = np.array([1,3,5,7])
D = np.subtract.outer(z, y)
print(D)
```

The output is the matrix of surpluses:

```
[[ 8  6  4  2]
 [ 5  3  1 -1]
 [ 3  1 -1 -3]
 [ 1 -1 -3 -5]]
```

This code block adds all the fictitious partners:

```
E = np.zeros((8,8))
E[0:4,0:4]=D
print(E)
```

The resulting matrix is:

```
[
[ 8.  6.  4.  2.  0.  0.  0.  0.]
 [ 5.  3.  1. -1.  0.  0.  0.  0.]
 [ 3.  1. -1. -3.  0.  0.  0.  0.]
 [ 1. -1. -3. -5.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

Now you can use a version of the Hungarian algorithm written by James Yu, an honors student here at the VSE at the time of writing.

```
from hungarianalg.alg import Hungarian as hg
result = hg(E)
print(result)
```

The result of the algorithm looks like this:

```
Matching:
['    ', '6.0', '    ', '    ', '    ', '    ', '    ', '    ', '    ']
['5.0', '    ', '    ', '    ', '    ', '    ', '    ', '    ', '    ']
['    ', '    ', '    ', '    ', '    ', '    ', '    ', '    ', '0.0']
['    ', '    ', '    ', '    ', '    ', '    ', '    ', '0.0', '    ']
['    ', '    ', '    ', '    ', '    ', '    ', '0.0', '    ', '    ']
['    ', '    ', '    ', '    ', '    ', '0.0', '    ', '    ', '    ']
['    ', '    ', '    ', '    ', '0.0', '    ', '    ', '    ', '    ']
['    ', '    ', '    ', '0.0', '    ', '    ', '    ', '    ', '    ']
['    ', '    ', '0.0', '    ', '    ', '    ', '    ', '    ', '    ']
Row Potentials: [5.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Column Potentials: [3.0, 1.0, 0, 0, 0, 0, 0, 0]
```

The matching is given in the table. The important parts are the row and column potentials that provide the surplus for each of the players. The buyers are represented by the rows, so buyer 1 should earn surplus 5, while buyer 2 should earn surplus 2. The other two buyers earn nothing because they aren't matched.

The columns represent the sellers, sellers 1 and 3 earn surplus 3 and 1 respectively. Notice that those surpluses are exactly the ones we would have predicted since our sellers' offer double auction would specify the trading price 4.

0.0.2. *Extra Proofs.*

Proposition. *If $\{E, \lambda, \mu\}$ is the equality graph, potential and matching produced by this algorithm. Then there is at least one column j for which $\lambda_j = 0$.*

Proof. (due to Alex Dong and James Yu) By (0.6), a column weight is adjusted during relabeling only if it is in $C(e)$ for some alternating path e. Since every alternating path starts in a free row, each such column is used in a partial matching μ contained in e. By Proposition , any column that is used in the partial matching is also used when an augmenting path is closed. Since an augmenting path associated with e must end in free column, it must therefore end in a column j' for which $\lambda_{j'} = 0$. The last column added to a complete matching must therefore be in a column with $\lambda_{j'} = 0$. □

0.0.3. *An example where more than one alternating path is used when relabelling.* Example (due to Callum McAllister) try 1:

150	126	125	150*	0
175	149	135	175*	0
250	122	148	250*+	0
160	139	140	160*	0

			26	
150	126	125	150	0
149	149*+	135	175*	0
224	122	148	250*+	0
160	139	140	160	0

alternating path $(2,3)\,(3,3)$

			26	
126	126*	125	150	0
149	149*+	135	175*	0
224	122	148	250*+	0
160	139	140	160	0

alternating path $(2,3)\,(2,1)$

	1		26	
125	126*	125*	150	0
148	149*+	135	175*	0
224	122	148	250*+	0
160	139	140	160	0

alternating path $(1,1)\,(2,1)(2,3)\,(3,3)$

	1		26	
125	126*	125*+	150	0
148	149*+	135	175*	0
224	122	148	250*+	0
160	139	140	160	0

close augmenting path

	1		26	
125	126*	125*+	150	0
148	149*+	135	175*	0
224	122	148	250*+	0
140	139	140*	160	0

no alternating path starts in a free row, relable row 4

	126	125	152	
0	126*	125*+	150	0
23	149*+	135	175*	0
98	122	148	250*+	0
15	139	140*	160	0

alternating path $(4,2)\,(1,2)\,(1,1)\,(2,1)\,(2,3)\,(3,3)$

	126	125	152	
0	126*	125*	150	0*+
23	149*+	135	175*	0
98	122	148	250*+	0
15	139	140*+	160	0

close augmenting path

Example try 2: Choose the other free row, the algorithm stalls

150	126	125	150*	0
175	149	135	175*	0
250	122	148	250*+	0
160	139	140	160*	0

			26	
150	126	125	150	0
149	149*+	135	175*	0
224	122	148	250*+	0
160	139	140	160	0

alternating path $(2,3)\,(3,3)$ - same as above

			26	
150	126	125	150	0
149	149*+	135	175*	0
224	122	148	250*+	0
140	139	140*+	160	0

relable free row 4 instead of doing 1, close the resulting augmenting path

			26	
126	126*	125	150	0
149	149*+	135	175*	0
224	122	148	250*+	0
140	139	140*+	160	0

now relable free row 1

	1		27	
125	126*	125*	150	0
148	149*+	135	175*	0
223	122	148	250*+	0
140	139	140*+	160	0

alternating path is $(1,1)\,(2,1)\,(2,3)\,(2,4)$,relable

neither of the alternating paths starting in row 1 is an augmenting path, but relabelling either as a single alternating path yields a δ value of 0. This is why the alternating paths must be relabelled at the same time.

When we combine the alternating paths all the columns are used so we have to check column 4 to look for the minimum delta. The delta values are just the row weights, giving $\delta = 125$. Relabelling gives

	126	125	152	
0	126*	125*	150	0*+
23	149*+	135	175*	0
98	122	148	250*+	0
15	139	140*+	160	0

close augmenting path. Same as above.