# Optimization Theory (DS2) Lecture #9 Minimum Cost Perfect Matching in Bipartite Graphs and the Hungarian Algorithm

December 6, 2016

**Abstract**

This week's particular form of shamanism is the Hungarian algorithm, which we will get to after starting by explaining bipartite graphs and the idea of minimum weight matching. We'll also look at a subroutine for finding a perfect matching (any one will do) on a subgraph of our original complete bipartite graph. The Hungarian algorithm can be explained using either matrix operations or graph operations; we'll try to reconcile the two. I'll try not to bring too much quantum computing into this, but you never know what might happen.

## 1 Bipartite Graphs

Take a graph $G = \{V, E\}$ where $V$ is the set of vertices (nodes) and $E$ is the set of edges (links). A few random facts about bipartite graphs:

1. A graph is *bipartite* if you can separate the nodes into two sets $U$ and $V$ where every edge crosses *between* the two sets, and no edges connect two nodes *within* the set.
2. Equivalently, the graph is *two-colorable*. (Question: Can you identify a condition which makes this true? Hint: What about cycles on the graph, and what about trees?)
3. A *matching* $M$ is a subset of the edges such that no node (vertex) has more than one edge in $M$.
4. A matching is *perfect* if every vertex has *exactly* one edge in $M$. (That is, a matching might not cover all of the vertices, but a perfect matching does.)

Many interesting problems can be mapped to bipartite graphs.

## 2 Mapping the Problem

Consider the following problem: the four people $A$, $B$, $C$ and $D$ are each offering to do jobs 1-4 for you, for the prices in the following table:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 80 & 40 & 50 & 46 \\ 40 & 70 & 20 & 25 \\ 30 & 10 & 20 & 30 \\ 35 & 20 & 25 & 30 \end{pmatrix} \end{array} \tag{1}$$

How can we assign the four people to the four jobs for the minimum cost? This can easily be mapped to a bipartite graph, with the nodes $A - D$ on the left and the nodes $1 - 4$ on the right, and an edge connecting each one on the left to each one on the right, in this case all with non-zero edge costs as in the matrix.

It can also be mapped to a linear program that is the relaxation of the equivalent integer problem:

$$\min z(\vec{x}) = \vec{c}^{\intercal} \vec{x} \tag{2}$$

subject to

$$A\vec{x} = \vec{1} \tag{3}$$

$$\vec{x} \geq 0 \tag{4}$$

$$\vec{x} \text{ integer.} \tag{5}$$

where

$$\vec{c} = \begin{pmatrix} 80 \\ 40 \\ 50 \\ 46 \\ 40 \\ 70 \\ 20 \\ 25 \\ 30 \\ 10 \\ 20 \\ 30 \\ 35 \\ 20 \\ 25 \\ 30 \end{pmatrix} \tag{6}$$

$$A = \begin{array}{c|cccccccccccccccc} & A1 & A2 & A3 & A4 & B1 & B2 & B3 & B4 & C1 & C2 & C3 & C4 & D1 & D2 & D3 & D4 \\ \hline A & 1 & 1 & 1 & 1 & & & & & & & & & & & & \\ B & & & & & 1 & 1 & 1 & 1 & & & & & & & & \\ C & & & & & & & & & 1 & 1 & 1 & 1 & & & & \\ D & & & & & & & & & & & & & 1 & 1 & 1 & 1 \\ 1 & 1 & & & & 1 & & & & 1 & & & & 1 & & & \\ 2 & & 1 & & & & 1 & & & & 1 & & & & 1 & & \\ 3 & & & 1 & & & & 1 & & & & 1 & & & & 1 & \\ 4 & & & & 1 & & & & 1 & & & & 1 & & & & 1 \end{array} \tag{7}$$

and the ordering of entries in $\vec{c}$ corresponds to the edges matching the labels across the top of $A$. Or, we can solve the dual problem:

$$\max z'(\vec{y}) = \vec{1}^\intercal \vec{y} \tag{8}$$

subject to

$$A^\intercal \vec{y} \leq \vec{c}. \tag{9}$$

However, there are other ways to solve this particular problem than via a linear (or integer) program. The textbook begins with a rather rigorous, but abstract, approach in order to prove some basic facts, but I found the description rather opaque. The pseudocode for that approach is in Algorithm 3.3, in a separate document.

The most famous method for solving the problem is the Hungarian algorithm. There are two descriptions of this algorithm, the matrix-oriented description and the graph description. In either case, we're going to need a way to find a perfect matching – *any* perfect matching, not necessarily a minimum cost one – so let's turn to that first.

## 3 Finding Some Random Perfect Matching

(Okay, not random in the sense of random number, just arbitrary, with no other important characteristics.)

During the course of the Hungarian algorithm as a whole, we're going to be given a subset of the edges in our original complete bipartite graph, and be asked if a perfect matching on that subset is possible. This subset will come to us in the form of a set of edges for which we have successfully negotiated our price differential from the minimum down to zero. (Don't get too excited, it doesn't mean people are going unpaid or that we're getting away for free, just that in the accounting process we've already set aside enough money for them, tracked somewhere else.)

### 3.1 Matrix Form

If this comes to us in matrix form, it will look something like:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} & 0 & & \\ & & 0 & 0 \\ & 0 & & \\ 0 & 0 & & \end{pmatrix} \end{array} \tag{10}$$

or for a slightly larger problem maybe something like

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} & 0 & & & & \\ & & 0 & 0 & & 0 \\ & & 0 & 0 & & 0 \\ 0 & 0 & & & & \\ & & 0 & & 0 & \\ & & 0 & & 0 & \end{pmatrix} \end{array} \tag{11}$$

where we don't care what the non-zero entries are (yet). Let's stick with the simpler one for the moment. In this form, we can pick matching elements using the following recipe:

1. Go through the rows, and if there is *exactly* one zero in the row, assign that job (that is, we're going to use the edge connecting the name on the left to the name at the top). Mark out all of the other zeros in the same column. (If there are two rows with exactly one zero in the same column, the choice of which to assign is arbitrary; we already know that there can be no perfect matching, but we'll do our best anyway. The algorithm will converge to the same solution, or an equivalently good one, eventually.)

2. Now do the equivalent with the columns, looking for columns that haven't yet been assigned and which have exactly one zero. Assign those, and mark out all of the other zeros in the same row. (Up to this point, it's a one-way street.)

   Xing out the ones we can't use and marking the ones we are using in angle brackets gives:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \left(\begin{array}{cccc} & <0> & & \\ & & 0 & 0 \\ & \cancel{0} & & \\ <0> & \cancel{0} & & \end{array}\right) \end{array} \tag{12}$$

3. Now what's left is rows and columns with multiple zeros each, and we're looking to see if there is a perfect matching. In this case, B has two entries left and C has none, so there is no perfect matching. For the moment, we will stop here.

The next step is to find the minimum number of horizontal and vertical lines that we can draw through the matrix to *cover all of the zeros*. There are various methods for doing this, including a fair amount of erroneous information on the web. The example on the English Wikipedia is confusing (again, I'd love it if somebody fixed that!). If that number is less than the number of rows in our column, we're not yet finished.

The minimum here is three lines, in column 2, row B, and row D.

This procedure (from
`http://math.stackexchange.com/questions/590305/finding-the-minimum-number-of-lin`
seems to work:

1. Mark all unassigned (unassignable) rows. (C)

2. Mark all previously unmarked columns that have zeros in the marked rows. (2)

3. Mark all rows that have *assigned* zeros in marked columns. (A)

4. Go back to step 2 unless there are no more columns that need to be marked.

(nothing new) This should give you:

$$\begin{array}{c} \\ A(X) \\ B \\ C(X) \\ D \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2(X) & 3 & 4 \end{array} \\ \left(\begin{array}{cccc} & <0> & & \\ & & 0 & 0 \\ & \cancel{0} & & \\ <0> & \cancel{0} & & \end{array}\right) \end{array} \tag{13}$$

5. Draw a line through every *marked* column and every *unmarked* row. (2, B, D)

This procedure should be the same as the one on Wikipedia, but is worded slightly more clearly. It's also the same as the procedure in Papadimitriou and Steiglitz, with some slight differences in notation.

Note that in this case, we might not have found a perfect matching even if one exists, because we stopped with some unallocated zeros still available. This simple procedure is only effective for the simplest cases. You can augment it by examining all possible combinations, but that's expensive. Instead, let's turn to the graph-based form of the perfect matching, which I think is both enlightening and useful.

## 3.2 The Graph Form

This graph-based approach to finding a perfect matching is the same as the one in Algorithm 3.4 in the textbook. The pseudocode is replicated in a separate document.

Let's pause to note that any tree of odd depth (counting the root as depth zero and its neighbors as depth one) is bipartite; simply begin with the root in your left hand group, and alternate placing the nodes into the right hand then left hand groups.

(An aside: Many trees can be colored to create a perfect matching. We're not going to directly use this fact, but a slightly different algorithm might; we are going to use similar characteristics of a tree, however.)

We begin with some matching, perhaps the empty set, $M = \varnothing$. We build a tree $T = (\{r\}, \varnothing)$ (the first argument to $T$ is the list of nodes, the second the list of edges) on the graph $G$, with the root $r$, such that $r$ is *M-exposed*, meaning that $r$ is *not* covered by an edge that is a member of $M$. From there, branch out and build a tree, using all of $r$'s neighbors, $N_G(\{r\})$. From there, keep branching, trying to build a spanning tree (as usual, avoiding links that make loops back to nodes we've already hit). The key is that *links at alternating depths should be part of the matching $M$*, such that each path back to the root is an *$M$-alternating path*.

At some point in building this tree, you should find another node, say $v$,that is *not* a member of the tree. Now you have a path from $v$ back to the root $r$ that has one link on the "leaf" side of the tree, a series of nodes that are part of $M$, to some node (say $s$) on the far side of the tree and then the root $r$ itself, which we started with outside of $M$. This gives us the opportunity to augment $M$. We do this by *flipping* the set of links along this path; the first and last links become new members of $M$ and everything in between flips polarity. This guarantees that the size of $M$ grows by one.

Now our root $r$ is part of $M$, so we need to select a new root $r'$ outside of $M$ and begin building a new tree $T$.

Pseudocode for this is in Algorithm 3.4 in the book, or in the separate document.

## 4 Hungarian Algorithm

Somewhat surprisingly, given that normally I have a strong preference for intuitive graph algorithms rather than heavyweight matrix algebra, in this case I find the matrix construction of the problem simpler. However, I'm going to try to integrate the two descriptions here, since it bugs me that they are often so different. The graph-based approach is the same as the one in Algorithm 3.5 in the textbook. The pseudocode is replicated in a separate document.

Fundamentally, what we're going to do is to tinker with our array of costs until some of them become zero (this doesn't change the outcome of the algorithm, for reasons explained below), then use those zero entries (which represent edge costs in our graph) to create a *subgraph*, and check to see if it's possible to make a perfect matching for the entire graph using only the edges in the subgraph. If not, rinse and repeat.

(Note: My organization into steps here doesn't exactly correspond to any of the sources below.)

**First step:** In our matrix above, we can see that, assuming we are going to assign any work to $A$, we must pay her at least forty dollars. Let's pull that forty dollars out of every matrix entry in row $A$, and just remember it off to the left:

$$\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} A(40) \\ B \\ C \\ D \end{array} & \begin{pmatrix} 40 & 0 & 10 & 6 \\ 40 & 70 & 20 & 25 \\ 30 & 10 & 20 & 30 \\ 35 & 20 & 25 & 30 \end{pmatrix} \end{array} \tag{14}$$

You can repeat this for all of the rows:

$$\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} A(40) \\ B(20) \\ C(10) \\ D(20) \end{array} & \begin{pmatrix} 40 & 0 & 10 & 6 \\ 20 & 50 & 0 & 5 \\ 20 & 0 & 10 & 20 \\ 15 & 0 & 5 & 10 \end{pmatrix} \end{array} \tag{15}$$

Summing those numbers to the left gives you 90. Dead minimum, we are going to wind up paying ninety bucks to get our four jobs done; the numbers now in the graph are the *extra* over \$90 (or *penalty*) we are going to pay. If we can find an assignment that matches a zero in every column and row, we get away at *exactly* \$90. In this case, there isn't one.

**Step Two:** So do the same thing with the columns, skipping columns that already have zeros in them:

$$
\begin{array}{c}
 \\ A(40) \\ B(20) \\ C(10) \\ D(20)
\end{array}
\begin{array}{c}
\begin{array}{cccc} 1(15) & 2 & 3 & 4(5) \end{array} \\
\left(\begin{array}{cccc}
25 & 0 & 10 & 1 \\
5 & 50 & 0 & 0 \\
5 & 0 & 10 & 15 \\
0 & 0 & 5 & 5
\end{array}\right)
\end{array}
\tag{16}
$$

For our minimum price, we add in that \$15 and \$5, and our minimum total is now \$110.

We now have at least one zero in every row and column, but may not be able to assign the jobs such that we hit one of those zeros for every assignment.

If you look at the anti-diagonal, it's all zeros except for a single 1. That assignment would give us a total cost, then, of \$111. Is that the minimum? More generally, assuming we're not quite there yet, how do we do a partial assignment and look for something better? How can we decide if there is or isn't a perfect assignment in our current arrangement?

**Steps Three and Four:** In some descriptions of the algorithm, this step and the next are combined. The goal in these two steps is to figure out if we have reached the point where an assignment (minimum cost perfect matching) is possible, and if so, to find that assignment. In some descriptions, only the *test* is done, after which the author/presenter says, "And now we know that an assignment is possible, so we're done!" and logs off. Here, we will attempt to find an assignment, then if that fails, use the information from the partial assignment to figure out our next step in the algorithm.

In the matrix approach, the approach to figuring out if an assignment is possible is to determine the minimum number of lines necessary to cross out all of the zeros in our array. We will defer that to Step Four.

So, **Step Three, matrix form**: Try to assign jobs so that the penalty is zero. If at any point in the following steps you have assigned exactly one zero in each column and each row, you are done, and the algorithm can terminate.

Take the zeros *only* from Eq. 16, and you'll have

$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D
\end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\left(\begin{array}{cccc}
 & 0 & & \\
 & & 0 & 0 \\
 & 0 & & \\
0 & 0 & &
\end{array}\right)
\end{array}
\tag{17}
$$

which is the matrix we ran the matching algorithm on back in Sec. 3.1.

**Step Four, matrix form:** Using the procedure above, find the minimum number of horizontal and vertical lines that we can draw through the matrix to *cover all of the zeros*. If that set is the size of the number of our nodes on each side, we are done.

**Steps Three and Four, graphical form:** Use the algorithm that finds a perfect matching on the graph, as discussed. This will return a matching, which might or might not be perfect, and a set of nodes for which it can't yet match. If the size of that latter set is zero, we're done.

**Step Five:** We are going to modify the weights of edges for which we don't have a matching, which is either those *uncovered* in the matrix, or left out of the matching in the graph form (the set $S = B(T)$ in the pseudocode). There are at least two ways to do this. Both begin by finding the smallest entry remaining in the matrix that does *not* have a line drawn through it. In the above example, this should be a 1, in the upper right hand corner.

Take that value, and do one of two things:

1. *Subtract* it from every entry in every *uncovered* row, and *add* it to every entry in every *covered* column; or

2. Subtract it only from the uncovered entries, then add it the entries covered *twice* (by a vertical and a horizontal line).

$$\begin{array}{c|cccc|} & 1(15) & 2 & 3 & 4(5) \\ A(40) & 24 & 0 & 9 & 0 \\ B(20) & 5 & 51 & 0 & 0 \\ C(10) & 4 & 0 & 9 & 14 \\ D(20) & 0 & 1 & 5 & 5 \end{array} \tag{18}$$

This will increase the number of zeros we have by one.

From here, return to Step Three, and try again to assign all of our tasks to see if we are finished. If not, continue the loop of Steps Three to Five, this sequence will eventually terminate. (In this example, it should actually terminate on the next loop, when we test for a perfect matching; fairly obviously, the anti-diagonal is a good assignment.)

---

## 5 Certificate

It is also possible to generate a certificate here that proves optimality (or the lack of a solution). We're going to skip that part.

## 6 Blossom V Algorithm

Quantum computing, glory be!

The Blossom V algorithm is commonly cited as a good way to solve the minimum weight perfect matching on a general graph (that is, not necessarily bipartite), and solving this problem is necessary for an important form of quantum error correction.

## 7 Source Material for This Lecture

Some text and examples adapted from Sec. 1.4.2 and Sec. 3.2 of the paperback edition of *A Gentle Introduction to Optimization*, B. Guerin *et al.* Example adapted from farnboroughmaths' YouTube video (showing the matrix-oriented interpretation of the Hungarian algorithm). I found Tim Roughgarden's CS261 lecture notes on minimum-cost bipartite matching (especially the history and the clear discussion of optimality proofs; he takes a graphical approach to the algorithm, rather than a matrix-oriented one) and Prashant Puaar's YouTube video (especially the discussion of how to mark zeros) to be helpful. The other main book I have been using, especially for graph algorithms, is Christos Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. It's a little dense, and way more material than we can cover in one semester, but it's a good book. There is also a short description of the problem and a different algorithm in Ch. 26 of Cormen *et al.* I found the English Wikipedia page to most helpful after I had begun to understand the algorithm, and of only minimal help in the beginning. I found the Harvard Math 20 Spring 2005 slides/notes (by Derek Bruff) to have simple, clear examples, but to not be of much use in the step of finding the minimum set of lines to cover the zeros. I have adopted their method for the addition step, as I find it simpler.

As of this writing, Kuhn's original paper, with a retrospective introduction by him, is available at
`http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Lec10-HungarianMethod-Ku`
This is from the book Jünger *et al.*, *50 Years of Integer Programming 1958–2008*.

There is also the website
`http://www.hungarianalgorithm.com/index.php`!

## 8 Conclusion

Sadly, this is the last full week of studying graph theory problems, but this is an extraordinarily rich area. I highly recommend you dig deeper into it.

Sources:

- Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, 3rd edition, Part VI, Chapters 22-26, about 160 pages of great material.
- Diestel. You'll find many books on the topic of graph theory, if you look, but this is the classic. (Japanese translation of 2nd edition is by Yagami's Ohta-sensei, but in English it's now up to 5th edition, available in Kindle now and hardback on Jan. 1, 2017)
- Another old classic, Bondy & Murty, is available for free online at `http://www.maths.lse.ac.uk/Personal/jozef/LTCC/Graph_Theory_Bondy_Murty.pdf`.
- `http://math.stackexchange.com/questions/27480/what-are-good-books-to-learn-g`
- I can offer up several papers on the topology of the Internet, and how it plays into various assessments of performance and robustness. Note that I am *not* a fan

of what are called “scale-free networks”; they can be used well, but in general that term is insufficiently specific and many papers using it are naive about their problem domain. (This is a little far afield from optimization, but you’ll enjoy it anyway.)

## 9 Homework

See the separate file uploaded to SFS.