211. Hungarian algorithm:
a Short Presentation using Sage

Sebastiano Ferraris
sebastiano.ferraris@gmail.com

Dedicated to Federica Narciso,
to her courageous sweetness in life
and her tenacious talent in Mathematics.

The following pages are the result of sincere but disconcerted passion, some hours of spare time and the help of Luca Lussardi, Nicola De Nitti and Giuliano "Nuanda" De Rossi.
This is to be intended as an introduction to the Hungarian Algorithm as well as a set of examples of recursive algorithm in Python. Any contribution, as suggestion, correction or improvement is welcome, so please do not hesitate to leave a comment or send me an email!

Introduction

Almost every article about Hungarian starts with the example of worker and tasks. According to the tradition, imagine to be the manager of a manufacturing company and to have to decide the allocation of 4 workers in 4 different workstations. Every worker has different experiences and characteristics and expresses a preference on the station in which he would like to work. The preference is expressed by giving a score from 1 to 20 (where 1 is the job that they prefer and 20 is the job they would like to avoid)
Your scope is to assign workers to tasks by making sure that everyone has a job close to its preference. A way to find a solution to this problem, without comparing manually all the possibilities, is called *Hungarian algorithm*. It was published by Harold Kuhn in 1955, grounded on previous works made by two Hungarian mathematicians, Denes Konig and Jeno Egervary [5]. In a more sophisticated perspective, not presented here, this problem can be seen as a particular application of the Hitchcock algorithm [3].

Bipartite Matching

A way to represent input data is using dots and connectors. Dots are used to represent both workers (Abigail, Baldwin, Cade, Diane) and tasks ($t1$, $t2$, $t3$, $t4$). Connectors to represent possible decisions
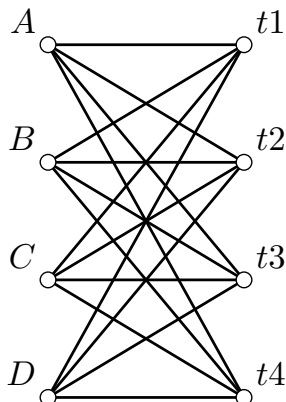
between workers and tasks. We assign to each connector a value corresponding to the score given by each worker to the connected task: if Abigail gives a score to the task $t1$ equals to 15, then the value of the connector from A to $t1$ will be 15.

Models made by dots and connectors are in general called *graphs*, dots are called *nodes* and connectors are called *edges*. Graph used in this situation are called *bipartite complete graph*. In the first partition we have workers and in the second partition we have tasks [1]. The number associated to each edge is called *weight of the edge.*

In general, given two sets A and B with the same number of elements $n = |A| = |B|$, for which is defined a *weight function w,*

$$w : A \times B \longrightarrow \mathbb{R}_+$$
$$(a, b) \longmapsto w(a, b)$$

such that for every couple (a, b) it associates a positive value, it is always possible to draw a corresponding bipartite complete graph [1]. A subset of edges with no common nodes is called (quite properly) *matching*. A matching in a bipartite complete graph with n nodes in each partition is *maximal* if you cannot add an edge to the matching while keeping the condition of not having common nodes for each edge. To each maximal matching corresponds a possible assignment workers-tasks.

In the figure below the maximal matching is highlighted in blue with the weight of the choose arch. Abigail is assigned to the job $t2$ (score 16), Baldwin to the $t3$ (score 2), Cade to the $t1$ (score 17) and Diane to $t4$ (score 14). The *weight of the matching* is the sum of the weights of the edges involved (in this case it is 49). Our aim is to find the maximal matching with minimal weight, i.e. to solve the so called *matching problem.*

---

[1]You may have to take your time to unpack this sentence. If X is a set, the number of its elements is written as $|X|$ and it is called cardinality of X. $A \times B$ is the cartesian product of the set A times the set B and it is the domain of the function w. The range of the function is the set of positive real numbers $\mathbb{R}_+$, the arrow $\mapsto$ means that the element (a, b) of the domain is mapped into $w(a, b)$.
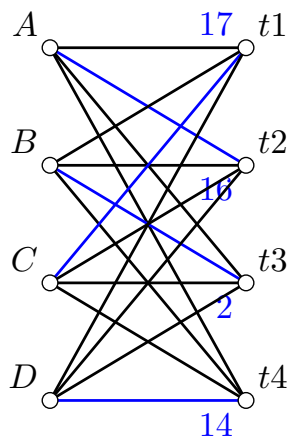
Cost Matrix

The representation of data with bipartite graph could be enough to reach the solution with a straight strategy, involving a device called *augmenting path*[2]. In the solution to the matching problem that is presented here, the bipartite graph is not enough, and a second tool is necessary: the *cost matrix*. Consider the set of employees as a vector in alphabetical order $\mathbf{e} = (A, B, C, D)$ and the vector of tasks ordered as $\mathbf{t} = (t1, t2, t3, t4)$; the *cost matrix* is defined as follows

$$M[i, j] = w(\mathbf{e}[i], \mathbf{t}[j])$$

Where $w(\mathbf{e}[i], \mathbf{t}[j])$ is the score that the worker $\mathbf{e}[i]$ gives to $\mathbf{t}[j]$.
In our leading example we will consider:

$$M = \begin{pmatrix} 15 & 16 & 19 & 4 \\ 6 & 1 & 2 & 6 \\ 17 & 3 & 5 & 12 \\ 13 & 6 & 16 & 14 \end{pmatrix} \tag{1}$$

At the position $[1, 0]$ of M we find[3] the score of Baldwin for the job $t1$, which is equal to 6. A set of n of elements of the cost matrix, such that none of them are on the same row or in the same vector corresponds to a matching. The association presented in the above figure can be represented in the cost matrix with bold numbers:

$$M = \begin{pmatrix} 15 & \mathbf{16} & 19 & 4 \\ 6 & 1 & \mathbf{2} & 6 \\ \mathbf{17} & 3 & 5 & 12 \\ 13 & 6 & 16 & \mathbf{14} \end{pmatrix}$$

---

[2]This way is well presented in [6].
[3]According to Python language, indexes start from zero: $M[1, 0]$ is the elment in the second row and in the firs column of M.

It is easy to see that the set of the feasible solutions, i.e. the possible associations, has 4! elements (4 possible on the first row, 3 on the second, 2 on the third). Each feasible solution can be represented in the most economical way with a vector **s** of dimension 4, such that in the first position we find the index of the column of the choose element in the first row of M. In the second position, the index of the column of the element in the second row of M and so on. In the previous case the vector corresponding to the matching is:

$$\mathbf{s} = (1, 2, 0, 3)$$

The *space of the feasible solution* $\mathcal{F}$ is the set of the vectors of dimension 4 containing numbers in the set $\{0, 1, 2, 3\}$ with no repetitions.
The *cost* of a vector **s** in $\mathcal{F}$ is defined as

$$w(\mathbf{s}) = \sum_{i=0}^{3} M[i, \mathbf{s}[i]]$$

and it corresponds to the weight of the matching.
The solution we are searching for is a vector in the set $\mathcal{F}$ with minimal cost (it is called *solution vector* and it may be not unique). Since $\mathcal{F}$ has $n!$ elements, it is inconvenient to employ a brute force algorithm to find it.
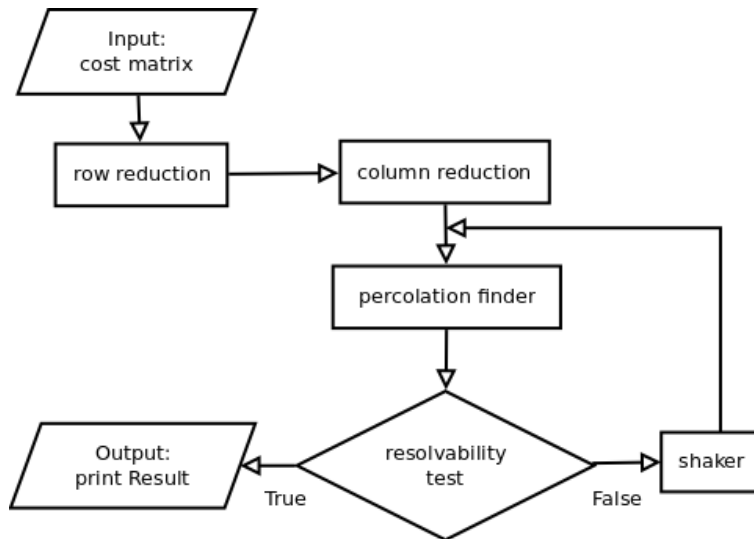

Figure 1: Hungarian Algorithm Flowchart

Step by Step Algorithm

We propose a version of the algorithm which uses 4 main functions (*row reduction, column reduction, percolation finder, shaker, resolvability test*), and 2 auxiliary function (*covering segments searcher*

and *redundance filter*). The *input* of the algorithm is the cost matrix M (it contains all information we need to solve the problem) and the *output* is the solution vector. In the following subsections each step is presented separately with Sage implementation and examples.

Step 1: Row Reduction

The step 1 consists of the application of a function which finds the minimal element to each row and subtract it to all the elements of the row.

Input: cost matrix M.

Output: M modified: for each row the minimal element is subtracted to each element of the row.

```
def row_reduction(M):
    '''
    row_reduction(M) \n
    step 1: row reduction
    '''
    S = copy(M)
    for i in range(S.nrows()):
        min_row = min([S[i,j] for j in range(S.ncols())])
        for j in range(S.ncols()):
            S[i,j] = S[i,j] - min_row
    return S
```

The matrix 1 is introduced in Sage as follows:

```
M = matrix([[15,16,19,4],[6,1,2,6],[17,3,5,12],[13,6,16,14]]); M
```

$$\begin{pmatrix} 15 & 16 & 19 & 4 \\ 6 & 1 & 2 & 6 \\ 17 & 3 & 5 & 12 \\ 13 & 6 & 16 & 14 \end{pmatrix}$$

```
M1 = row_reduction(M); M1
```

$$\begin{pmatrix} 11 & 12 & 15 & 0 \\ 5 & 0 & 1 & 5 \\ 14 & 0 & 2 & 9 \\ 7 & 0 & 10 & 8 \end{pmatrix}$$

Step 2: Column Reduction

Step 2 refers to the columns: for each column it finds the minimal element and subtract it to all of the elements of the column.

Input: cost matrix M.

Output: M modified: for each column the minimal element is subtracted to each element of the column.

```
def col_reduction(M):
    '''
    col_reduction(M) \n
    step 2: column reduction
    '''
    S = copy(M)
    for j in range(S.ncols()):
        min_col = min([S[i,j] for i in range(S.nrows())])
        for i in range(S.nrows()):
            S[i,j] = S[i,j] - min_col
    return S
```

Applying this in our leading example we get:

```
M2 = col_reduction(M1); M2
```

$$
\begin{pmatrix}
6 & 12 & 14 & 0 \\
0 & 0 & 0 & 5 \\
9 & 0 & 1 & 9 \\
2 & 0 & 9 & 8
\end{pmatrix}
\tag{2}
$$

The solution vector does not change if we consider M or M after having applied row or column reduction:

Lemma 1 *Let $\mathbf{s}_1$ and $\mathbf{s}_2$ two vectors in the space of feasible solutions, w the weight function defining M and $\hat{w}$ the weight function defining row_reduction(M). If $w(\mathbf{s}_1) \leq w(\mathbf{s}_2)$ then $\hat{w}(\mathbf{s}_1) \leq \hat{w}(\mathbf{s}_2)$.*

Proof: Let m_i the min in the i-th row. Subtract m_i to the i-th row is the same as subtract it to each edge with a common node of the first partition in the related bipartite complete graph. If w_i is the weight function defined by the matrix where we have subtracted the m_i only in the i-th row we have $w_i(\mathbf{s}) = w(\mathbf{s}) - m_i$.
It follows that $\hat{w}(\mathbf{s}) = w(\mathbf{s}) - \sum_{i=0}^{n-1} m_i$ for each $\mathbf{s}$ in the feasible space. $\square$

It can be shown that the lemma is still valid if $\hat{w}$ is the weight function of *col_reduction(M)*. It follows then:

Theorem 1 *The solution vector for the matrix M is the same as for the matrix col_reduction(row_reduction(M)) as well as for the matrix row_reduction(col_reduction(M)).*

Note that in general *col_reduction(row_reduction(M))* $\neq$ *row_reduction(col_reduction(M))*. Which means that you may not get the same matrix if the order of application of step 1 and step 2 is inverted.

Step 3: Percolation Finder

After step 1 and 2 we have got a matrix with some zeros (at least one for each row and one for each column). If it is possible to "walk" the matrix from the first row to the last just stepping on zeros and never passing twice on the same column, our walk will be the solution vector we are searching for. For example in the following matrix

$$N = \begin{pmatrix} 6 & 11 & 0 & \mathbf{0} \\ 12 & \mathbf{0} & 0 & 4 \\ \mathbf{0} & 15 & 5 & 1 \\ 8 & 14 & \mathbf{0} & 1 \end{pmatrix}$$

we can see such kind of walk and we can notice that the solution is $\mathbf{s} = (3, 1, 0, 2)$. In the matrix (2) a possible the walk through zeros corresponds to the vector $(3, 2, 1, 1)$ which is not a feasible solution. This rough idea of walk can be formalized under the name of percolation:

Definition 1 *Given a costs matrix M, a* **percolation** *over M is a subset of n element of the matrix M, one and only one for each row. If all the elements of a percolation are zeros, it is called* 0-**percolation**.

In general not all the matrices admits 0-percolation, but it easy to prove that every output of step 1 and step 2 admits at least one.

As made in our example we can identify every 0-percolation with a vector $\mathbf{v} = (j_1, j_2, \ldots, j_n)$ of dimension n made of integer in the set $\{0, 1, 2, \ldots, n-1\}$. With this nomenclature if $\mathbf{v}[i] = j_i$ then the element $M[i, j_i]$ belongs to the percolation. For example in the matrix N, 4 0-percolations are possible:

$$\begin{pmatrix} 6 & 11 & \mathbf{0} & 0 \\ 12 & \mathbf{0} & 0 & 4 \\ \mathbf{0} & 15 & 5 & 1 \\ 8 & 14 & \mathbf{0} & 1 \end{pmatrix} \quad \begin{pmatrix} 6 & 11 & 0 & \mathbf{0} \\ 12 & \mathbf{0} & 0 & 4 \\ \mathbf{0} & 15 & 5 & 1 \\ 8 & 14 & \mathbf{0} & 1 \end{pmatrix} \quad \begin{pmatrix} 6 & 11 & \mathbf{0} & 0 \\ 12 & 0 & \mathbf{0} & 4 \\ \mathbf{0} & 15 & 5 & 1 \\ 8 & 14 & \mathbf{0} & 1 \end{pmatrix} \quad \begin{pmatrix} 6 & 11 & 0 & \mathbf{0} \\ 12 & 0 & \mathbf{0} & 4 \\ \mathbf{0} & 15 & 5 & 1 \\ 8 & 14 & \mathbf{0} & 1 \end{pmatrix}$$

We are interested in see how many zeros of the same 0-percolation are in the same column, since the solution we are searching for will be the 0-percolation having 1 zero in each column. Clearly the second 0-percolation in four above listed is the the one we are interested in. In consequence of that we need a mathematical tool that states how far is our percolation from the one that we are searching:

Definition 2 *Given a 0-percolation* $\mathbf{v}$ *over M of length n, the index of redundancy of* $\mathbf{v}$ *is a real number defined as the maximal number of zeros of the percolation in the same column minus 1, plus the number of column which not posses any element of the percolation divided by the length of the percolation.*

Sometimes a formula can make a definition easier to be understand. This is one of this cases:

$$index\_of\_redundancy(\mathbf{v}) = max - 1 + max/n$$

where max is the maximal number of zeros of the percolation in the same column.
In the previous example we have

$(2,1,0,2)$	$(3,1,0,2)$	$(2,2,0,2)$	$(3,2,0,2)$
1.25	0	2.5	1.25

where each index of redundancy is shown under its percolation. Percolations with redundancy index
equals to 0 are called here *solution*. Given the percolation V a function to calculate the redundancy
index is

```
def redundance_index(V):
    n = len(V)
    occurrence_vector = [V.count(i) for i in range(n)]
    return max(occurrence_vector) -1 + occurrence_vector.count(0)*1.0/n
```

where *occurrence vector* at the position i gives the number of occurrence of the number i in the
vector V. If M admits at leas one 0-percolation with redundancy index equals to 0, a recursive[4]
algorithm to find it could be:
Input: cost matrix M
Output: True if the matrix admits a 0-percolation with redundancy index equals to 0, False other-
wise. In the global variable *ans* is recorded the first 0-percolation found if it exists.

```
ans = []

def percolation_finder_0(M):
    '''
    0-percolation finder
    '''
    def find_walk(M, row, sol):
        global ans
        for i in range(M.nrows()):
            if M[row,i] == 0 and i not in sol:
                sol = sol + [i]
                ans = sol
                # print ans
                if row >= M.nrows() - 1 or find_walk(M, row+1, sol):
                    return True
                else:
                    sol = sol[0:-1]
        return False

    if find_walk(M, 0, []):
        return True
    else:
        return False
```

---

[4]An introduction to recursive algorithm can be found on [2] as well as on wikipedia.

Uncommenting the line *print ans* it will be possible to see the recursive algorithm at work. Note that to use again this algorithm you must initialize the global variable to the empty list. If the matrix have more than one solution the previous algorithm gives us only the first that he finds. In general the matrix could admit no solution at all. In this case all the 0-percolation have redundancy index grater then 0.

This means that we have identified the following two cases:

1. $row\_reduction(col\_reduction(M))$ admits 0-percolation with redundancy index equals to 0.

2. $row\_reduction(col\_reduction(M))$ do not admits 0-percolation with redundancy index equals to 0.

and since we do not want to rely too much on luck, we need an algorithm that provides all the 0-percolation and tells if some of them has index of redundancy equals to zero or not. We use again recursion:

Input: cost matrix M.

Output: list of all of the 0-percolation.

```
walks = []

def percolation_finder_1(M):
    '''
    percolation finder vers 1
    '''
    def scout(M, breadcrumbs):
        global walks
        nrows = M.nrows()
        ncols = M.ncols()
        crumbs_number = len(breadcrumbs)
        #print breadcrumbs
        if crumbs_number < ncols:
            for j in range(ncols):
                if M[crumbs_number,j] == 0:
                    scout(M,breadcrumbs + [j])
        elif crumbs_number == ncols:
            walks = walks + [breadcrumbs]

    for j in range(M.ncols()):
        if M[0,j] == 0:
            scout(M,[j])
    return walks
```

As before, uncommenting the line *print breadcrumbs*, we can see recursion at work[5]. You may have noticed that a matrix with a lots of zeros makes the walks list too big to be manageable. Since we

---

[5]You may be not familiar with global and local variables in Python; if so take a look at [7] pag. 310.

are not interested in all the percolation but only in those with minimal redundancy, we propose a way to decide if we may want to stop after having collected "enough" percolations in the vector *walks*:

Input: cost matrix M, false if we do not want to find all the percolation, max number of percolation to be found.

Output: M again, followed by a list build as the list of all percolation followed by the redundace index.

```
def percolation_finder(M, find_all = True, max_percolation = 6):
    '''
    step 3 - percolation finder
    '''
    def redundance_index(V):
        n = len(V)
        occurrence_vector = [V.count(i) for i in range(n)]
        return max(occurrence_vector) -1 + occurrence_vector.count(0)*1.0/n

    def scout(M, breadcrumbs):
        global walks
        nrows = M.nrows()
        ncols = M.ncols()
        crumbs_number = len(breadcrumbs)
        #print breadcrumbs
        if crumbs_number < ncols:
            col_index_arranged = \\
            list(set(range(ncols))-set(breadcrumbs)) + list(set(breadcrumbs))
            for j in col_index_arranged:
                if M[crumbs_number,j] == 0:
                    scout(M, breadcrumbs + [j])
        elif crumbs_number == ncols:
            walks_recorder(breadcrumbs)

    def walks_recorder(V,find_all = True):
        global walks
        ri = redundance_index(V)
        if find_all:
            walks = walks + [V] + [ri]
        elif not find_all:
            if 0 not in walks and len(walks) < max_percolation:
                walks = walks + [V] + [ri]

    for j in range(M.ncols()):
        if M[0,j] == 0:
            scout(M,[j])
```

```
    return [M,walks]
```

The function *walks_recorder* adds to the global variable *walks* each new 0-percolation followed by its redundance index *ri*. If the variable *find_all*, in the input value, is set to *False* then the vector walks will collect only a number of 0-percolation equals to *max_percolation*.[6] It does make sense to stop the recursion before all 0-percolation are collected, just because the indexes of the first cycle-for had been rearranged:

```
col_index_arranged = list(set(range(ncols))-set(breadcrumbs)) + list(set(breadcrumbs))
for j in col_index_arranged:
```

This means that when the algorithm is arrived at the row *crumbs_number* it searches for a new zero before in columns that are not already been used. For example, if the input matrix is like the following (the best case to find a solution as well as one of the worst case for a recursive algorithm)

$$N = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

the first 0-percolation that the algorithm finds is the solution $(0, 1, 2, 3)$. So it makes sense to limit the solution, even thought if the number *max_percolation* is too little we may lose some multiple solution.
Applying this to our main example (matrix 2) we get:

```
walks
[[3, 0, 1, 1], 1.25, [3, 1, 1, 1], 2.5, [3, 2, 1, 1], 1.25]
```

Step 4: Resolvability Test

At this stage we have got the matrix reduced and the list of 0-percolation (complete or not) with related redundancy indexes. After the step 4 we will have the list of 0-percolation with minimal redundancy index, and a flag which states if the 0-percolation (or percolations) founded is already our solution or not.
Input: cost matrix M, list of 0-percolation, where each 0-percolation is followed by its index of redundancy.
Output: M again, followed by the list of 0-percolation with minimal index of redundancy, and with a flag, True if the minimal index is 0 and so we have already our solution, False otherwise.

```
def resolvability_test(M,walks):
    '''
    step 4: Resolvability Test
    '''
```

---

[6]Note that this just stop the recording, and do not stops the recursive steps. It is also possible to stop the recursion when the *max _percolation* is reached. We did not proposed it here for clarity, but it can be considered as an exercise.

```
    min_redundance = min(walks)
    filtered_walks = \\
    [walks[i] for i in range(len(walks))[::2] if walks[i+1] == min_redundance]
    if min_redundance == 0:
        flag = True
    else:
        flag = False
    return [M, filtered_walks, flag]
```

The instance $range(len(walks))[::2]$ is the list of the redundancy index in the vector walks (all the odd position in the vector).

Let see how does it work on our main example:

```
walks = []
A,B = percolation_finder(M2); A,B
resolvability_test(A,B)
```

$$\left[\begin{pmatrix} 6 & 12 & 14 & 0 \\ 0 & 0 & 0 & 5 \\ 9 & 0 & 1 & 9 \\ 2 & 0 & 9 & 8 \end{pmatrix}, [3,0,1,1], [3,2,1,1], \text{False}\right] \tag{3}$$

When the flag is True the job is done, and we can print the solution. If it is False we must do some manipulation on the matrix in order to add some zeros without change the results. This tricky passage is here the step number 5, called shaker.

Step 5: Shaker

To perform this passage, first of all we must find the *covering segments* of the input matrix M. A covering of M is the minimal set of horizontal or vertical segments that covers all the zeros in the matrix. In [4], it is suggested to actually draw horizontal or vertical lines over the matrix, covering all the zeros such that the number of lines is minimal. For reasons related to the implementation of the algorithm we do not follow that path, even if it is more intuitive (and is therefore strongly recommended as a starting point of view). Instead of using covering segments, we use a couple of vectors called *covered_col* and *covered_row*:

Definition 3 *Let M be a square matrix with real entries of dimension $n \times n$. We define two vectors of dimension n called* covered_col *and* covered_row *with elements in the set $\{0, 1\}$ such that if $M[i, j]$ is a zero then or* covered_row$[i] = 1$ *or* covered_col$[j] = 1$ *and the number of 1 in both vectors is minimal.*

The non-zero elements in the vector covered_row are called horizontal covering segments. *The non-zero elements in the vector covered_col are called* vertical covering segments.

For the matrix 2 those vectors are:

$$covered\_row = [0, 1, 0, 0] \qquad covered\_col = [0, 1, 0, 1]$$

$$
\begin{array}{c}
\begin{array}{cccc}
\mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1}
\end{array} \\
\begin{array}{c}
\mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0}
\end{array}
\begin{pmatrix}
6 & 12 & 14 & 0 \\
0 & 0 & 0 & 5 \\
9 & 0 & 1 & 9 \\
2 & 0 & 9 & 8
\end{pmatrix}
\end{array}
$$

The algorithm that we use to find covering segments requires the matrix as well as one 0-percolation with minimal redundancy index. It is called *covering_segments _searcher* and will be used as auxiliary function of the shaker:

Input: cost matrix M, one of the 0-percolation with minimal redundancy index.
Output: position of horizontal and vertical covering segments.

```
def covering_segments_searcher(M, min_redundance_percolation):
    '''
    step 5, auxiliary function
    '''
    # (A)
    nrows = M.nrows()
    ncols = M.ncols()
    marked_row = [0]*nrows
    marked_col = [0]*ncols
    # (B)
    occurrence_vector = [min_redundance_percolation.count(i) for i in range(nrows)]
    for pos in range(nrows):
        if occurrence_vector[pos] > 1:
            duplicates_pos = \\
            [k for k in range(nrows) if min_redundance_percolation[k] == pos][1:]
            for j in duplicates_pos:
                marked_row[j] = 1
    # (C)
    flag_mark = 1
    while flag_mark != 0 :
        flag_mark = 0
        # (C-1)
        for i in range(nrows):
            if marked_row[i] == 1:
                for j in range(ncols):
                    if M[i,j] == 0 and marked_col[j] != 1:
                        marked_col[j] = 1
                        flag_mark = flag_mark + 1
        # (C-2)
        for j in range(ncols):
            if marked_col[j] == 1:
                for i in range(nrows):
```

```
            if M[i,j] == 0 and marked_row[i] != 1
            and min_redundance_percolation[i] == j:
                marked_row[i] = 1
                flag_mark = flag_mark + 1
    # (D)
    covered_row = [(i+1)%2 for i in marked_row]
    covered_col = marked_col

    return (covered_row, covered_col)
```

It performs the following tasks:

(A) Initialize marked_row and marked_col as zero vectors. If we want to mark a the row i or a column j we substitute a 1 at the i position of marked _row or at the j position of marked_col.

(B) Mark all the row that possess a zero that makes the input 0-percolation redundant (i.e. if $(M)_{h,j}$ and $(M)_{k,j}$ are zeros in the same column we mark the row k). The vector duplicates_pos gives the position of all the duplicate zeros in the same columns.

(C-1) Mark all the column having a zero in the row marked in the previous step.

(C-2) Mark all the row having zeros in marked cols, only if those zeros are in the 0-percolation with minimal redundancy (so if their cols are in the min_redundant_percolation vector).

(C) Perform (C-1) and (C-2) in sequence until it is possible to mark new positions.

(D) Vertical covering segments are the segments marked in previous step in the column, whereas horizontal covering segments are all the unmarked row.

It can be proved that covering_segments_searcher finds the covering segments for the input matrix. Moreover the number of covering segments is always equals or less than the dimension of the matrix. In case of it is equals to its dimension, the vector min_redundant_percolation has redundancy index equals to 0. The concept of covering segments leads us to the crucial definition of *uncovered*, *covered* and *bi-covered* elements in the matrix M.

Definition 4 *Let M be a square matrix with real entries of dimension $n \times n$ considered with* covered_col *and* covered_row *vectors.*
If at the element $M[i,j]$ corresponds covered_row$[i] = 0$ and covered_col$[j] = 0$ then the element $M[i,j]$ is called uncovered *(and certainly is not a zero).*
If only one between covered_row$[i]$ and covered_col$[j]$ is equals to zero then the element $M[i,j]$ is called covered.
If both covered_row$[i]$ and covered_col$[j]$ are equals to 1 then the element $M[i,j]$ is called bi-covered.
The total number of non-zero entries in covered_col *and* covered_row *is called* covering number[7].

---

[7]It can be defined a discrete monotonous function g, such that (covering number) + g(redundancy index) = n.

Now we have all that we need to "shake" our matrix, which means perform what the following theorem states:

Theorem 2 *Let M be a matrix considered with* covered_col *and* covered_row *vectors and with a solution vector* **s** *(element of the space of feasible solution that minimize the cost of the association). Let m be the minimal element of the set of the uncovered elements. If m is subtracted to all the uncovered elements and 2m is added to all the bi-covered elements, the new matrix obtained in this way still admits* **s** *as solution vector.*

The matrix that results applying the previous theorem is called here *shaked matrix*. Our shaker algorithm is as follows:
Input: cost matrix M, set of walks with minimal redundance index.
Output: shaked matrix of M.

 (A) Uses auxiliary function to find the covering segments

 (B) Finds m the minimal uncovered element value.

 (C) m is subtracted to every uncovered element and $2m$ is added to each bicovered element.

```
def shaker(M, filtered_walks):
    '''
    step 5 - shaker
    '''
    nrows = M.nrows()
    ncols = M.ncols()
    min_redundance_percolation = filtered_walks[0]
    # (1)
    [cov_row, cov_col] = covering_segments_searcher(M, min_redundance_percolation)
    # (2)
    zeropos_in_cov_row = [i for i in range(nrows) if cov_row[i] == 0]
    zeropos_in_cov_col = [j for j in range(ncols) if cov_col[j] == 0]
    m = min([M[i,j] for i in zeropos_in_cov_row  for j in zeropos_in_cov_col])
    # (3)
    for i in range(nrows) :
        for j in range(ncols):
            if cov_row[i] == 0 == cov_col[j]:
                M[i,j] = M[i,j] - m
            elif cov_row[i] == 1 == cov_col[j]:
                M[i,j] = M[i,j] + 2*m
    return M
```

According to the flow diagram of figure 1, this step is performed after percolation finder and resolvability test until the resolvability test is not passed.
For the main example we have

```
walks = []
A,B = percolation_finder(M2); A,B
N = shaker(A,B); N
```

$$N = \begin{pmatrix} 6 & 14 & 14 & 0 \\ 0 & 2 & 0 & 5 \\ 8 & 0 & 0 & 8 \\ 1 & 0 & 8 & 7 \end{pmatrix} \tag{4}$$

Which admits a 0-percolation with redundancy index equals to 0 (it is our solution!). Applying again percolation finder and resolvability test we get

```
S = percolation_finder(N);
T = resolvability_test(S[0],S[1]); T
```

$$\left[\begin{pmatrix} 6 & 14 & 14 & 0 \\ 0 & 2 & 0 & 5 \\ 8 & 0 & 0 & 8 \\ 1 & 0 & 8 & 7 \end{pmatrix}, [[3, 0, 2, 1]], \text{True} \right] \tag{5}$$

An so the solution to the main problem (that is the association of workers to different tasks that maximizes satisfaction and productivity) is $[3, 0, 2, 1]$.

Why does it work?

In this section we want to answer the following questions:

1. Why does the solution vector remain unchanged after the shaker is applied to M?

2. Why does the shaker take us away from trouble in a finite number of steps?

The answer to the first question consists is the proof of theorem 2. We approach the proof via some lemmas:

Lemma 2 *Given a cost matrix M of dimension $n \times n$, if p is a percolation with redundancy index equals to 0, it is uniquely defined by a bijection σ between the set of indexes $\{0, 1, \dots, n-1\}$.*

Proof: A percolation $\mathbf{p} = (j_0, j_1, \dots, j_{n-1})$ with redundancy index equals to 0 is an vector that associate to each position (row in the corresponding matrix) one and only one integer number between 0 and $n-1$ (column in the corresponding matrix) without repetitions:

$$(j_0, j_1, \dots, j_{n-1}) \qquad \longmapsto \qquad \begin{pmatrix} 0 & 1 & 2 & \dots & n-1 \\ j_0 & j_1 & j_2 & \dots & j_{n-1} \end{pmatrix} = \sigma$$

Each percolation with redundancy index equals to 0 can be represented by the set of elements of the matrix $\{M[i, \sigma(i)]\}_{i=0}^{n-1}$. $\square$

Lemma 3 *Let M be a cost matrix of dimension $n \times n$ and* covering_row *and* covering_col *having $n-1$ non-zero elements (covering segments are n-1), and let σ a bijection of n elements. It must always exists an integer $a \in \{0, 1, \dots, n-1\}$ such that $M[a, \sigma(a)]$ is uncovered. Moreover if for some k, $M[k, \sigma(k)]$ is a bi-covered element then it must exists b and c such that $M[b, \sigma(b)]$ and $M[c, \sigma(c)]$ are uncovered.*

Proof: Such k do not always exists: all covering segments can be parallel. In this case either *covering_row* has only one zero and *covering_col* is all made by zeros or *covering_col* has only one zero and *covering_row* is all made by zeros. If we consider *covering_row* with only one zero at the position t then our uncovered element will be $M[t, \sigma(t)]$ and so we have proved the first part of the thesis for $a = t$.

Let $M[k, \sigma(k)]$ a bi-covered element, then we must have at least one non-zero element in *covering_col* in the position k and at least two zeros in the vector *covering_col*, in the position $a \neq k$ and $b \neq k$ (or vice versa one non-zero element in *covering_col* in the position $\sigma^{-1}(k)$ and at least two zeros in the vector *covering_row*).

If the percolation defined by σ, passes trough the bi-covered element (and therefore through $M[k, \sigma(k)]$), when it passes through the rows of M, indicated with a and b, it will necessarily encounter two uncovered elements, since $\sigma(a) \neq \sigma(k)$ and $\sigma(b) \neq \sigma(k)$. □

As a consequence of the previous lemma we have:

Lemma 4 *Let $\mathbf{s}_1$ and $\mathbf{s}_2$ be two vector in the space of the feasible solution, w the weight function of M and $\hat{w}$ the weight function defined by the shaked matrix of M. If $w(\mathbf{s}_1) \leq w(\mathbf{s}_2)$ then $\hat{w}(\mathbf{s}_1) \leq \hat{w}(\mathbf{s}_2)$.*

And now, considering the fact that The thesis of theorem 2 is equivalent to the lemma 4, theorem 2 has been proved.

The answer to the second questions is a consequence of the following results:

Lemma 5 *Let $M[i, j] = 0$ a bi-covered element in the cost matrix M. It follows that*

$$z\_cols[j] > 1$$
$$z\_rows[i] > 1$$

where $z\_cols[j]$ is the number of zeros in the col j of M and $z\_rows[i]$ is the number of zeros in the row i.

Proof: By contradiction, if $z\_rows[i] = 1$, there is only one zero on the i-th row. This must be $M[i, j]$, but it is already covered by the covering segment $z\_cols[j]$. According to the minimality of the covering segments we should have $z\_rows[i] = 0$ and $M[i, j] = 0$ should not be a bi-covered element. □

Every time step 4 is performed, we add a zero to the set of the uncovered elements, and we remove (in the worst case) a number of zero equals to the number of bi-covered elements. However according to the previous lemma this step do not reduces the non-zero element on the vectors z_rows and z_cols.

Since after step 1 and step 2 we have at least $n+1$ non-zero elements in both z_rows and z_cols, after a finite number of application of the steps 4, in the worst case given by

$$\sum_{i|z\_rows[i]>1} z\_rows[i] + \sum_{j|z\_cols[j]>1} z\_cols[j] + n - 1$$

all the zeros in vectors z_rows and z_cols we will be eliminated .

If there is no more zeros means that for each row and col we can always find a percolation with redundancy index equals to 0 and so we can be sure that step_4 take us away from trouble in a finite number of steps.

Hungarian Algorithm

Now it is possible to join all the previous steps in one single function, according to the flowchart presented at the beginning of the section.

```
def hungarian(M, find_all = True, max_percolation = 6):
    '''
    Hungarian Algorithm
    '''
    global walks
    walks = []
    cont = 0
    max_loop = max(M.nrows(), M.ncols())
    S = row_reduction(col_reduction(M))
    [S, walks] = percolation_finder(S, find_all, max_percolation)
    [S, filtered_walks, flag] = resolvability_test(S, walks)
    while flag == False and cont < max_loop:
        S = shaker(S, filtered_walks)
        walks = []
        [S, walks] = percolation_finder(S, find_all, max_percolation)
        [S, filtered_walks, flag] = resolvability_test(S, walks)
        cont = cont + 1
    walks = []
    return filtered_walks
```

Conclusion

A tool for finding the solution vector of any square matrix has been presented. Saving all functions here presented in a single one .py file we can load into Sage with command *load* and reload in the same session (if some modifications has happened) with the command *attach*:

```
load('/home/paht_to_file/filename.py')
attach('/home/paht_to_file/filename.py')
```

We can test it on a random matrix A

```
A = random_matrix(ZZ, 10, x = 0, y = 30); A
```

$$\begin{pmatrix} 18 & 3 & 39 & 27 & 2 & 4 & 34 & 8 & 0 & 5 \\ 11 & 0 & 11 & 17 & 23 & 25 & 32 & 22 & 14 & 0 \\ 0 & 13 & 0 & 12 & 16 & 12 & 0 & 11 & 14 & 1 \\ 8 & 28 & 10 & 0 & 24 & 0 & 19 & 3 & 17 & 18 \\ 16 & 17 & 36 & 0 & 0 & 26 & 32 & 5 & 3 & 20 \\ 7 & 11 & 11 & 7 & 21 & 4 & 0 & 18 & 8 & 13 \\ 23 & 14 & 26 & 8 & 2 & 15 & 26 & 0 & 24 & 10 \\ 11 & 3 & 12 & 0 & 25 & 0 & 24 & 0 & 8 & 1 \\ 3 & 1 & 0 & 7 & 5 & 12 & 22 & 10 & 13 & 10 \\ 0 & 0 & 7 & 7 & 0 & 23 & 25 & 17 & 15 & 14 \end{pmatrix}$$

```
s = hungarian(A); s
```

$$[[8, 9, 0, 3, 4, 6, 7, 5, 2, 1], [8, 9, 0, 5, 4, 6, 7, 3, 2, 1]]$$

If there are more employees than workstations, we can still use the Hungarian algorithm, adding dummy nodes to the second partition of the biparted complete graph. Each worker will give them a score from 1 to 20. 1 if they wish to find new job and be moved to another department or workplace; 20 if they prefer not to be moved.

Coming back to our leading example, we suppose that $t4 =$ "be move to another department" and the workers (A, B, C, D) are very loyal to their place so they give all a score equals to 20 for the node $t4$. The cost matrix will be:

$$M = \begin{pmatrix} 15 & 16 & 19 & 20 \\ 6 & 1 & 2 & 20 \\ 17 & 3 & 5 & 20 \\ 13 & 6 & 16 & 20 \end{pmatrix} \tag{6}$$

The solution is to move Abigail since

```
s = hungarian(M); s
```

$$[3, 0, 2, 1]$$

References

[1] J.A. Bondy, U. S. R. Murty, *Graph theory with applications*, Elevier Science Publishing, 1982.

[2] R. Sedgewick, P. Flajolet, *An introduction to Analysis of Algorithm*, Pearson, 1996.

[3] C. Papadmitriou, K. Steiglitz, *Combinatorial Optimization*, Drover Books, 1998.

[4] `http://www.hungarianalgorithm.com/`

[5] `http://en.wikipedia.org/wiki/Hungarian_algorithm`

[6] Mordecai J. Golin, *Bipartite Matching and the Hungarian Method*, Course Notes, Hong Kong University of Science and Technology.

[7] Mark Lutz, *Programming Python*, 3rd edition, Oreilly And Associates 2007.