Algorithm description

In the following code the applied algorithm is ‘Hungarian algorithm’.

Step by step description relating the algorithm’s steps with the applied code:

Imported packages:

```
import copy
import numpy as np
from ortools.graph.python import linear_sum_assignment
from scipy.optimize import linear_sum_assignment
import networkx as nx
from networkx.algorithms import bipartite
import numpy as np
import matplotlib.pyplot as plt
```

Function(1):

Get-min(mat)

=> ‘Get_minimum’ function is a function that takes the required matrix as an argument.

=> The objective of this function is:

1- loop over each row of the matrix and get the minimum number for each and subtract it from all the elements of the current row.

```
{ for row_num in range(mat.shape[0]):
    mat[row_num] = mat[row_num] - np.min(mat[row_num])
}
```

2- Loop over the columns of the matrix, for elements after being subtracted and get the minimum element in each column and subtract it from the current column.

```
{for col_num in range(mat.shape[1]):
    mat[:,col_num] = mat[:,col_num] - np.min(mat[:,col_num])
}
```

At last this function returns the new matrix.

```
{return mat
}
```

Function(2):

'**min_zero(zer-mat, mark-zero)**'

=> The objective of this function is to mark the minimum number of zeros in each row and column. It's a loop that will keep repeating until there are no more minimum number of zeros are found.

Steps:

1. The function takes two parameters (zero_mat), which is the matrix represents zeros (mark_zero), which is a list whose aim is to store the position AKA: column/row indices of the marked found zeros.

2. These variables will be intended to track the minimum number of zeros and their index

```
3. min_row_index = -1
4. zero_index = -1
```

3. The for loop is to Iterate over each row in the 'zero_mat' matrix to get row index and the row itself.

4. Since it's a boolean matrix, calculate the number of zeros in each current row by counting 'True' occurrences

```
num_zeros = np.sum(row == True)
```

5. The next condition is for checking if the number of zeros captures is greater than zero and less than the minimum number of the CURRENT number of zeros since we want the least of the minimum.

```
if num_zeros > 0 and num_zeros < min_sum:
   min_sum = num_zeros
   min_row_index = row_num
   zero_index = np.where(row == True)[0][0]
```

If the condition passed, add current row index and the index of the of the first occurrence of zero

6. After the loop there is another condition that checks if there are any entries assigned to 'min_row_index'. Meaning if it checked it and it was true therefore a minimum number of zero was found AND will be appending the , 'min_row_index' and 'zero_index' to 'mark_zero' list
   Else, if it's still -1 no entries for minimum zero found.

```
if min_row_index != -1:
   mark_zero.append((min_row_index, zero_index))
   zero_mat[min_row_index, :] = False
   zero_mat[:, zero_index] = False
```

By false it indicates that the row where zeros where marked set it to false meaning that all zeros in that place were captured. Same goes for columns.

Function(3):

'**mark_matrix(mat)**'

=> The objective of the function is that it iterates over the given parameter matrix through rows, and columns to get the positions of each of marked zeros, rows, and columns.

The following function returns 3 lists, 'marked_zero', 'marked_row', and 'marked_cols' that represent the position of marked/canceled zeros, rows, and cols.

Steps:

1. 'marked_zero'-> to store positions of marked zeros.
   'marked_row'-> Follows/ store marked rows.
   'marked_cols'->Follow/store marked columns.

```
    marked_zero = []
    zero_row = []
    zero_col = []
```

2. The parameter that will be sent to this function is a matrix. We will make a copy of it. This copy will be a similar formula but a boolean matrix of trues and false (True=zeros, False=any other value).

```
current_mat = mat
zero_bool_mat = (current_mat==0)
zero_bool_mat_copy = zero_bool_mat.copy()
```

3. Next up is a while loop that loops as long as it finds (True, AKA: zeros) in the ‘zero_bool_mat_copy’ capturing zeros by calling the previous ‘min_zero’ function, which finds minimum number of zero index of either row or column and appends it in the ‘marked_zero’ list.

```
while(True in zero_bool_mat_copy):

   min_zero(zero_bool_mat_copy, marked_zero)
```

4. To sort positions out, to identify the rows index and columns index another two lists were made which are ‘zero_row’ &’zero_column’.

```
for i in range (len(marked_zero)):

   zero_row.append(marked_zero[i][0])
   zero_col.append(marked_zero[i][1])
```

N.B: [0] means rows & [1] means columns.

5. ‘Non_marked_row’ list will contain elements of all the possible remaining values of the matrix by subtracting the row indices at which zeros were marked from the rest indices of other values.

```
non_marked_row = list(set(range(current_mat.shape[0])) - set(zero_row))
```

6. By constructing ‘marked_cols’ list and and adding check boolean variable and set it to True.
   The next loop will keep looping over the ‘non_marked_row’ indices list and access each row element to check if there are columns of available zero entries to mark it out.
   A condition will check if the found row element value is True=zero, and its column index AKA: j is not already in the ‘marked_cols’ list, append that column index and reset the check boolean to True.

```
marked_cols = []
check = True
while check:

   check = False

   for i in range(len(non_marked_row)):
       row_array = zero_bool_mat[non_marked_row[i],:]
       for j in range(row_array.shape[0]):
           if(row_array[j]==True and j not in marked_cols):
               marked_cols.append(j)
               check = True
```

7. Another for loop that will iterate over the ‘marked_zero’ list to check if the current row_num/ current position is not in the ‘non_marked_row’ list AKA: that row is marked then. Same goes for columns.

Then subtract non_marked_row list elements from the rest of elements. And set the check back to true.

```
for row_num, col_num in marked_zero:

       if row_num not in non_marked_row and col_num in marked_cols:

           non_marked_row.append(row_num)
           check_switch = True

marked_rows = list(set(range(mat.shape[0])) - set(non_marked_row))
return (marked_zero, marked_rows, marked_cols)
```

Function(4):

‘**final_matrix(mat,cover_rows, cover_cols)**’

=> The objective of this function is to apply the last touches/steps in the algorithm by subtracting the minimum value from all non-covered (AKA: values that don’t have any lines passing through them) and adding the same value to covered cells (AKA: values that have intersection of both horizontal and vertical lines).

Steps:

1. Make a copy from the matrix like previously done to make sure nothing will affect the main one.
2. Two lists are constructed ‘non_covered_rows’, and ‘non_covered_cols’, they represent rows and columns not present in ‘cover_rows’ and ‘cover_cols’ lists.
3. Iterate over non_covered_rows and cols to collect non_zero elements in the non_covered cells of the matrix.
4. Find minimum value among collected elements and assign to min_num variable.

```
cur_mat = mat.copy()

non_covered_rows = [row for row in range(len(cur_mat)) if row not in
cover_rows]
non_covered_cols = [col for col in range(len(cur_mat[0])) if col not in
cover_cols]

non_zero_elements = [cur_mat[row][col] for row in non_covered_rows for
col in non_covered_cols]

min_num = min(non_zero_elements)
```

5. Iterate over non_covered rows and columns , both of each indices subtract min_num from the corresponding element in current_mat.
6. Then iterate over covered rows and columns and do the same previous step but adding min_num to covered ones.

```
for row in non_covered_rows:
   for col in non_covered_cols:
       cur_mat[row, col] -= min_num

for row in cover_rows:
   for col in cover_cols:
       cur_mat[row, col] += min_num

return cur_mat
```

Function(5):

'**Hungarian_integration**' which is a wrap up for all previous functions integrating them together.

=> The objective of this function is applying fully the hungarian algorithm starting by minimizing till adjusting the matrix to find an optimal assignment of rows and columns with minimum cost.

Steps:

1. N indicates the number of the square matrix (AKA: number of rows)
2. Copy matrix as previously explained.
3. Call Get_min(mat) to get the minimum number as previously explained.
4. Count_zero_lines is a variable that represents row or column in the matrix where zero is assigned
5. While loop iterates applying the algorithm until all lines are found.
6. Call mark_matrix that marks zeros in matrix as previously explained.
7. Calculate new value of Count_zero_lines by adding rows and columns lengths.
8. The condition checks if new Count_zero_lines is still < n meaning there are still rows/columns that contain zero. If true, the final_matrix function will be called to adjust the matrix based on marked rows and columns as previously explained.

```
def hungarian_integration(cost_matrix):

    n = cost_matrix.shape[0]
    cur_mat = copy.deepcopy(cost_matrix)

    cur_mat = Get_min(cost_matrix)

    count_zero_lines = 0

    while count_zero_lines < n:

        ans_pos, marked_rows, marked_cols = mark_matrix(cur_mat)
        count_zero_lines = len(marked_rows) + len(marked_cols)

        if count_zero_lines < n:

            cur_mat = final_matrix(cur_mat, marked_rows,marked_cols)

    return ans_pos
```

Function(6):

‘**min_cost(mat.pos)**’

=> The objective of this function is to calculate the total cost by summing the values of mat the given positions which are the best assignment position AKA: minimum cost and creates a new matrix that contains the values of these positions only.

```
def min_cost(mat,pos):
   total = 0
   ans_mat = np.zeros((mat.shape[0],mat.shape[1]))
   for i in range(len(pos)):
       total+=mat[pos[i][0], pos[i][1]]
       ans_mat[pos[i][0],pos[i][1]] = mat[pos[i][0], pos[i][1]]

   return total,ans_mat
```

Applying GUI for this problem by using networkx, bipartite:

```
B = nx.DiGraph()

top_nodes = [1,2,3]
bottom_nodes = [4,5,6]

B.add_nodes_from(top_nodes,bipartite=0,color='#C5E0B4')
B.add_nodes_from(bottom_nodes,bipartite=1,color='#FFE699')

node_pos = nx.bipartite_layout(B,top_nodes)

B.add_edge(1,4,weight=19,color='b',width=1)
B.add_edge(1,5,weight=28,color='b',width=1)
B.add_edge(1,6,weight=31,color='b',width=1)

B.add_edge(2,4,weight=11,color='b',width=1)
B.add_edge(2,5,weight=17,color='b',width=1)
B.add_edge(2,6,weight=16,color='b',width=1)

B.add_edge(3,4,weight=12,color='b',width=1)
B.add_edge(3,5,weight=15,color='b',width=1)
B.add_edge(3,6,weight=13,color='b',width=1)
```

```
my_matching =
bipartite.matching.minimum_weight_full_matching(B,top_nodes,"weight")
# print(my_matching)

assignments = list(my_matching.items())
edge_colors = ["r" if edge in assignments else '#C4C2C6' for edge in B.edges()]
edge_width = [5 if edge in assignments else 1 for edge in B.edges()]


node_colors = list(nx.get_node_attributes(B,'color').values())
nx.draw(B, pos=node_pos, with_labels=True, font_color='red', node_size=1000,
node_color=node_colors, edge_color=edge_colors, width=edge_width)
label1=nx.get_edge_attributes(B,'weight')
nx.draw_networkx_edge_labels(B,node_pos,edge_labels=label1, label_pos=0.85)
# plt.show()
```

Simply identifying n x n for the desired matrix AKA: indices and for each edge identify the index from worker to the specified job and the weight/cost of it.