Assignment 6
Of
Modelling & Simulation Lab (CS1052)

Masters of Technology in Computer Science And Engineering

submitted by
Arghya Bandyopadhyay
RollNo. 20CS4103

submitted to
Dr Nanda Dulal Jana
Assistant Professor
Dept. of CSE

National Institute of Technology, Durgapur

Problem Statement :

There are four jobs to be assigned to five similar machines. Only one job can be assigned to one machine. The amount of time in hour required for the jobs per machines are given in the following matrix

Machines

Jobs	A	B	C	D	E
1	4	3	6	2	7
2	10	12	11	14	16
3	4	3	2	1	5
4	8	7	6	9	6

Find an optimal assignment of jobs to the machines to minimize the total processing time and also find out for which machine no jobs is assigned. What is the total processing time to complete all the jobs.

Problem formulat^n -

checking for Balanced or unbalanced

no of Rows = 4 and no of column = 5

Since, no of rows ≠ no of column

so, unbalanced assignment problem.

we add dummy rows to balance.

	A	B	C	D	E
1	$\overset{x_{11}}{4}$	$\overset{x_{12}}{3}$	$\overset{x_{13}}{6}$	$\overset{x_{14}}{2}$	$\overset{x_{15}}{7}$
2	$\overset{x_{21}}{10}$	$\overset{x_{22}}{12}$	$\overset{x_{23}}{11}$	$\overset{x_{24}}{14}$	$\overset{x_{25}}{16}$
3	$\overset{x_{31}}{4}$	$\overset{x_{32}}{3}$	$\overset{x_{33}}{2}$	$\overset{x_{34}}{1}$	$\overset{x_{35}}{5}$
4	$\overset{x_{41}}{8}$	$\overset{x_{42}}{7}$	$\overset{x_{43}}{6}$	$\overset{x_{44}}{9}$	$\overset{x_{45}}{5}$
D5	$\overset{x_{51}}{0}$	$\overset{x_{52}}{0}$	$\overset{x_{53}}{0}$	$\overset{x_{54}}{0}$	$\overset{x_{55}}{0}$

$$\text{obj } \min(z) = \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ij}$$

$$= 4x_{11} + 3x_{12} + 6x_{13} + 2x_{14} + 7x_{15} +$$
$$10x_{21} + 12x_{22} + 11x_{23} + 14x_{24} + 16x_{25}$$
$$+ 4x_{31} + 3x_{32} + 2x_{33} + 1x_{34} + 5x_{35}$$
$$+ 8x_{41} + 7x_{42} + 6x_{43} + \cancel{x_{44}} + 9x_{44}$$
$$+ 5x_{45}$$

$$\begin{array}{c|ccccc} & A & B & \overset{\Downarrow}{C} & D & E \\ \hline 1 & 2 & 1 & 4 & 0 & 5 \\ 2 & 0 & 2 & 1 & 4 & 6 \\ 3 & 3 & 2 & 1 & 0 & 4 \\ 4 & 2 & 1 & 0 & 3 & 0 \\ D5 & 0 & 0 & 0 & 0 & 0 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 0 & 0 & 0 & 0 & 0 \end{array}$$

There is no column reduction possible as min of each column is '0'

(2) Remove zero by using min no of horizontal & vertical lines.

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline 1 & 2 & 1 & 4 & 0 & 5 \\ 2 & 0 & 2 & 1 & 4 & 6 \\ 3 & 3 & 2 & 1 & 0 & 4 \\ 4 & 2 & 1 & 0 & 3 & 0 \\ D5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

no of lines = 4, which is less than order of matrix i.e $4 < 5$

So, we choose min from the uncovered element from the matrix & add it to intersected point and substract from uncovered leave the rest as it is

Step 3

	A	B	C	D	E
1	2	0	3	0	4
2	0	1	0	4	5
3	3	1	0	0	3
4	3	1	0	4	0
D5	1	0	0	1	0

again repeat the step 2 & 3 until no. of lines is equal to order of matrix i.e n == N

Step 4

	A	B	C	D	E
1	2	0	3	0	4
2	0	1	0	4	5
3	3	1	0	0	3
4	3	1	0	4	0
D5	1	0	0	1	0

Here we can see that no of lines = 5 which is equal to order of matrix i.e 5 = = 5, so we move to '0' marking step.

(5)

	A	B	C	D	E
1	2	0	3	$\cancel{0}$	4
2	0	1	$\cancel{0}$	4	5
3	3	1	$\cancel{0}$	0	3
4	3	1	0	4	$\cancel{0}$
D_5	1	$\cancel{0}$	$\cancel{0}$	1	0

So from the above table,

→ Job 1 is assigned to machine B
→ cost 3

→ Job 2 assigned to machine A
→ cost 10

→ Job 3 is assigned to mach. D
→ cost = 1

→ Job 4 is assigned to machi. C
→ cost 6

→ Dummy job row is assign. to E. Hence E is ideal.

Min total processing time = 3+10+1+6
= 20

Python Code:

```
class HungarianError(Exception):
    pass

try:
    import numpy as np
except ImportError:
    raise HungarianError("NumPy is not installed.")


class Hungarian:
    """
    Implementation of the Hungarian (Munkres) Algorithm using np.
    Usage:
        hungarian = Hungarian(cost_matrix)
        hungarian.calculate()
    or
        hungarian = Hungarian()
        hungarian.calculate(cost_matrix)
    Handle Profit matrix:
        hungarian = Hungarian(profit_matrix, is_profit_matrix=True)
    or
        cost_matrix = Hungarian.make_cost_matrix(profit_matrix)
    The matrix will be automatically padded if it is not square.
    For that numpy's resize function is used, which automatically adds 0's to any row/column that
    is added
    Get results and total potential after calculation:
        hungarian.get_results()
        hungarian.get_total_potential()
    """

    def __init__(self, input_matrix=None, is_profit_matrix=False):
        """
        input_matrix is a List of Lists.
        input_matrix is assumed to be a cost matrix unless is_profit_matrix is True.
        """
        if input_matrix is not None:
            # Save input
            my_matrix = np.array(input_matrix)
            self._input_matrix = np.array(input_matrix)
            self._maxColumn = my_matrix.shape[1]
            self._maxRow = my_matrix.shape[0]

            # Adds 0s if any columns/rows are added. Otherwise stays unaltered
            matrix_size = max(self._maxColumn, self._maxRow)
            pad_columns = matrix_size - self._maxRow
            pad_rows = matrix_size - self._maxColumn
            my_matrix = np.pad(my_matrix, ((0, pad_columns), (0, pad_rows)), 'constant',
  constant_values=(0))

            # Convert matrix to profit matrix if necessary
            if is_profit_matrix:
                my_matrix = self.make_cost_matrix(my_matrix)

            self._cost_matrix = my_matrix
            self._size = len(my_matrix)
            self._shape = my_matrix.shape

            # Results from algorithm.
            self._results = []
            self._totalPotential = 0
        else:
            self._cost_matrix = None
```

```
    def get_results(self):
        """Get results after calculation."""
        return self._results

    def get_total_potential(self):
        """Returns expected value after calculation."""
        return self._totalPotential

    def calculate(self, input_matrix=None, is_profit_matrix=False):
        """
        Implementation of the Hungarian (Munkres) Algorithm.
        input_matrix is a List of Lists.
        input_matrix is assumed to be a cost matrix unless is_profit_matrix is True.
        """
        # Handle invalid and new matrix inputs.
        if input_matrix is None and self._cost_matrix is None:
            raise HungarianError("Invalid input")
        elif input_matrix is not None:
            self.__init__(input_matrix, is_profit_matrix)

        result_matrix = self._cost_matrix.copy()

        # Step 1: Subtract row mins from each row.
        for index, row in enumerate(result_matrix):
            result_matrix[index] -= row.min()

        # Step 2: Subtract column mins from each column.
        for index, column in enumerate(result_matrix.T):
            result_matrix[:, index] -= column.min()

        # Step 3: Use minimum number of lines to cover all zeros in the matrix.
        # If the total covered rows+columns is not equal to the matrix size then adjust matrix
and repeat.
        total_covered = 0
        while total_covered < self._size:
            # Find minimum number of lines to cover all zeros in the matrix and find total
covered rows and columns.
            cover_zeros = CoverZeros(result_matrix)
            covered_rows = cover_zeros.get_covered_rows()
            covered_columns = cover_zeros.get_covered_columns()
            total_covered = len(covered_rows) + len(covered_columns)

            # if the total covered rows+columns is not equal to the matrix size then adjust it by
 min uncovered num (m).
            if total_covered < self._size:
                result_matrix = self._adjust_matrix_by_min_uncovered_num(result_matrix,
covered_rows, covered_columns)

        # Step 4: Starting with the top row, work your way downwards as you make assignments.
        # Find single zeros in rows or columns.
        # Add them to final result and remove them and their associated row/column from the
matrix.
        expected_results = min(self._maxColumn, self._maxRow)
        zero_locations = (result_matrix == 0)
        while len(self._results) != expected_results:

            # If number of zeros in the matrix is zero before finding all the results then an
error has occurred.
            if not zero_locations.any():
                raise HungarianError("Unable to find results. Algorithm has failed.")

            # Find results and mark rows and columns for deletion
            matched_rows, matched_columns = self.__find_matches(zero_locations)

            # Make arbitrary selection
            total_matched = len(matched_rows) + len(matched_columns)
```

```
            if total_matched == 0:
                matched_rows, matched_columns = self.select_arbitrary_match(zero_locations)

            # Delete rows and columns
            for row in matched_rows:
                zero_locations[row] = False
            for column in matched_columns:
                zero_locations[:, column] = False

            # Save Results
            self.__set_results(zip(matched_rows, matched_columns))

        # Calculate total potential
        value = 0
        for row, column in self._results:
            value += self._input_matrix[row, column]
        self._totalPotential = value

    @staticmethod
    def make_cost_matrix(profit_matrix):
        """
        Converts a profit matrix into a cost matrix.
        Expects NumPy objects as input.
        """
        # subtract profit matrix from a matrix made of the max value of the profit matrix
        matrix_shape = profit_matrix.shape
        offset_matrix = np.ones(matrix_shape, dtype=int) * profit_matrix.max()
        cost_matrix = offset_matrix - profit_matrix
        return cost_matrix

    def _adjust_matrix_by_min_uncovered_num(self, result_matrix, covered_rows, covered_columns):
        """Subtract m from every uncovered number and add m to every element covered with two
lines."""
        # Calculate minimum uncovered number (m)
        elements = []
        for row_index, row in enumerate(result_matrix):
            if row_index not in covered_rows:
                for index, element in enumerate(row):
                    if index not in covered_columns:
                        elements.append(element)
        min_uncovered_num = min(elements)

        # Add m to every covered element
        adjusted_matrix = result_matrix
        for row in covered_rows:
            adjusted_matrix[row] += min_uncovered_num
        for column in covered_columns:
            adjusted_matrix[:, column] += min_uncovered_num

        # Subtract m from every element
        m_matrix = np.ones(self._shape, dtype=int) * min_uncovered_num
        adjusted_matrix -= m_matrix

        return adjusted_matrix

    def __find_matches(self, zero_locations):
        """Returns rows and columns with matches in them."""
        marked_rows = np.array([], dtype=int)
        marked_columns = np.array([], dtype=int)

        # Mark rows and columns with matches
        # Iterate over rows
        for index, row in enumerate(zero_locations):
            row_index = np.array([index])
            if np.sum(row) == 1:
                column_index, = np.where(row)
```

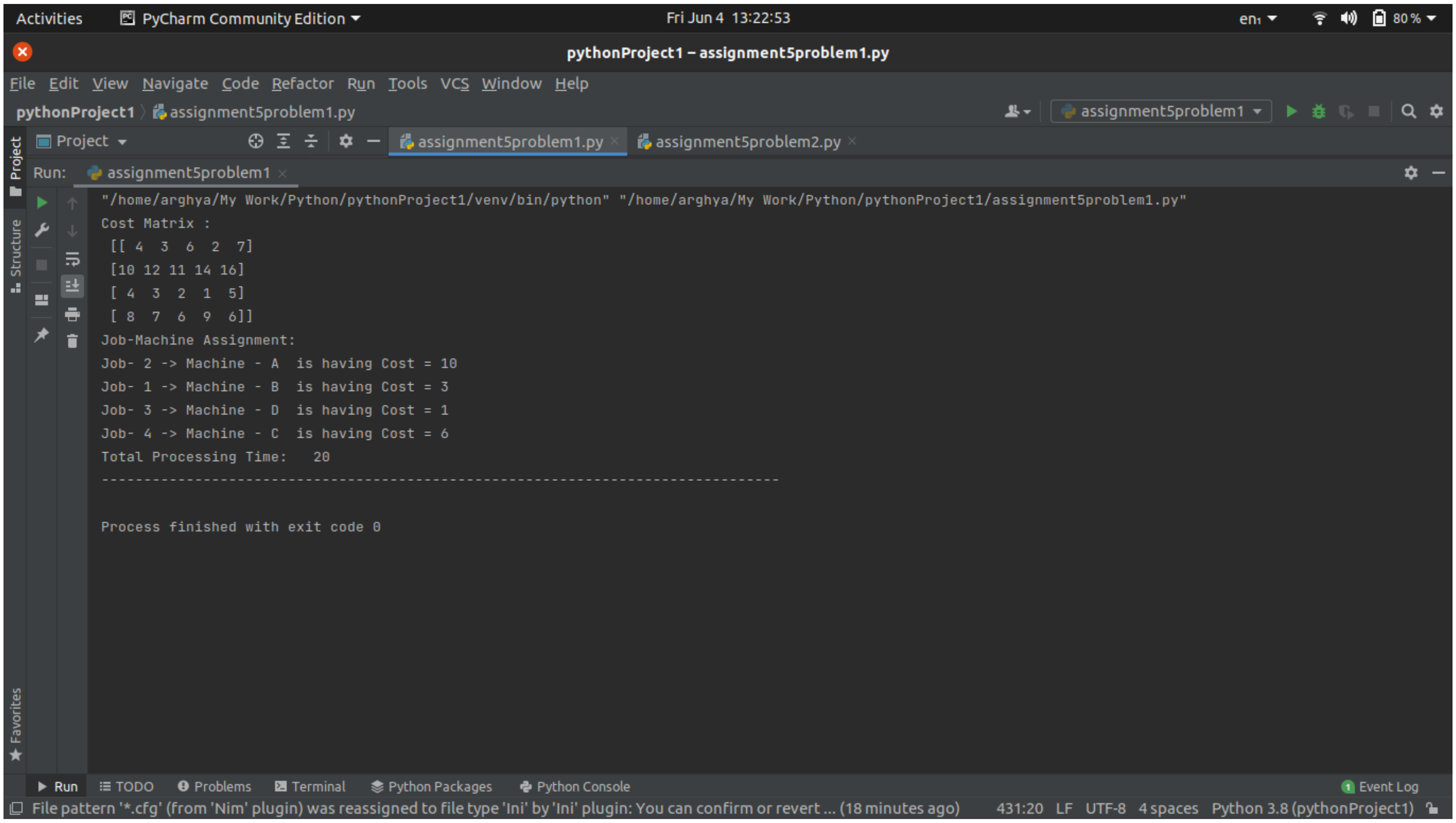
```
                marked_rows, marked_columns = self.__mark_rows_and_columns(marked_rows,
marked_columns, row_index,
                                                                            column_index)

        # Iterate over columns
        for index, column in enumerate(zero_locations.T):
            column_index = np.array([index])
            if np.sum(column) == 1:
                row_index, = np.where(column)
                marked_rows, marked_columns = self.__mark_rows_and_columns(marked_rows,
marked_columns, row_index,
                                                                            column_index)

        return marked_rows, marked_columns

    @staticmethod
    def __mark_rows_and_columns(marked_rows, marked_columns, row_index, column_index):
        """Check if column or row is marked. If not marked then mark it."""
        new_marked_rows = marked_rows
        new_marked_columns = marked_columns
        if not (marked_rows == row_index).any() and not (marked_columns == column_index).any():
            new_marked_rows = np.insert(marked_rows, len(marked_rows), row_index)
            new_marked_columns = np.insert(marked_columns, len(marked_columns), column_index)
        return new_marked_rows, new_marked_columns

    @staticmethod
    def select_arbitrary_match(zero_locations):
        """Selects row column combination with minimum number of zeros in it."""
        # Count number of zeros in row and column combinations
        rows, columns = np.where(zero_locations)
        zero_count = []
        for index, row in enumerate(rows):
            total_zeros = np.sum(zero_locations[row]) + np.sum(zero_locations[:, columns[index]])
            zero_count.append(total_zeros)

        # Get the row column combination with the minimum number of zeros.
        indices = zero_count.index(min(zero_count))
        row = np.array([rows[indices]])
        column = np.array([columns[indices]])

        return row, column

    def __set_results(self, result_lists):
        """Set results during calculation."""
        # Check if results values are out of bound from input matrix (because of matrix being
padded).
        # Add results to results list.
        for result in result_lists:
            row, column = result
            if row < self._maxRow and column < self._maxColumn:
                new_result = (int(row), int(column))
                self._results.append(new_result)


class CoverZeros:
    """
    Use minimum number of lines to cover all zeros in the matrix.
    Algorithm based on: http://weber.ucsd.edu/~vcrawfor/hungar.pdf
    """

    def __init__(self, matrix):
        """
        Input a matrix and save it as a boolean matrix to designate zero locations.
        Run calculation procedure to generate results.
        """
        # Find zeros in matrix
```

```python
        self._zero_locations = (matrix == 0)
        self._shape = matrix.shape

        # Choices starts without any choices made.
        self._choices = np.zeros(self._shape, dtype=bool)

        self._marked_rows = []
        self._marked_columns = []

        # marks rows and columns
        self.__calculate()

        # Draw lines through all unmarked rows and all marked columns.
        self._covered_rows = list(set(range(self._shape[0])) - set(self._marked_rows))
        self._covered_columns = self._marked_columns

    def get_covered_rows(self):
        """Return list of covered rows."""
        return self._covered_rows

    def get_covered_columns(self):
        """Return list of covered columns."""
        return self._covered_columns

    def __calculate(self):
        """
        Calculates minimum number of lines necessary to cover all zeros in a matrix.
        Algorithm based on: http://weber.ucsd.edu/~vcrawfor/hungar.pdf
        """
        while True:
            # Erase all marks.
            self._marked_rows = []
            self._marked_columns = []

            # Mark all rows in which no choice has been made.
            for index, row in enumerate(self._choices):
                if not row.any():
                    self._marked_rows.append(index)

            # If no marked rows then finish.
            if not self._marked_rows:
                return True

            # Mark all columns not already marked which have zeros in marked rows.
            num_marked_columns = self.__mark_new_columns_with_zeros_in_marked_rows()

            # If no new marked columns then finish.
            if num_marked_columns == 0:
                return True

            # While there is some choice in every marked column.
            while self.__choice_in_all_marked_columns():
                # Some Choice in every marked column.

                # Mark all rows not already marked which have choices in marked columns.
                num_marked_rows = self.__mark_new_rows_with_choices_in_marked_columns()

                # If no new marks then Finish.
                if num_marked_rows == 0:
                    return True

                # Mark all columns not already marked which have zeros in marked rows.
                num_marked_columns = self.__mark_new_columns_with_zeros_in_marked_rows()

                # If no new marked columns then finish.
                if num_marked_columns == 0:
```

```
                return True

        # No choice in one or more marked columns.
        # Find a marked column that does not have a choice.
        choice_column_index = self.__find_marked_column_without_choice()

        while choice_column_index is not None:
            # Find a zero in the column indexed that does not have a row with a choice.
            choice_row_index = self.__find_row_without_choice(choice_column_index)

            # Check if an available row was found.
            new_choice_column_index = None
            if choice_row_index is None:
                # Find a good row to accomodate swap. Find its column pair.
                choice_row_index, new_choice_column_index = self.
__find_best_choice_row_and_new_column(
                    choice_column_index)

                # Delete old choice.
                self._choices[choice_row_index, new_choice_column_index] = False

            # Set zero to choice.
            self._choices[choice_row_index, choice_column_index] = True

            # Loop again if choice is added to a row with a choice already in it.
            choice_column_index = new_choice_column_index

    def __mark_new_columns_with_zeros_in_marked_rows(self):
        """Mark all columns not already marked which have zeros in marked rows."""
        num_marked_columns = 0
        for index, column in enumerate(self._zero_locations.T):
            if index not in self._marked_columns:
                if column.any():
                    row_indices, = np.where(column)
                    zeros_in_marked_rows = (set(self._marked_rows) & set(row_indices)) != set([])
                    if zeros_in_marked_rows:
                        self._marked_columns.append(index)
                        num_marked_columns += 1
        return num_marked_columns

    def __mark_new_rows_with_choices_in_marked_columns(self):
        """Mark all rows not already marked which have choices in marked columns."""
        num_marked_rows = 0
        for index, row in enumerate(self._choices):
            if index not in self._marked_rows:
                if row.any():
                    column_index, = np.where(row)
                    if column_index in self._marked_columns:
                        self._marked_rows.append(index)
                        num_marked_rows += 1
        return num_marked_rows

    def __choice_in_all_marked_columns(self):
        """Return Boolean True if there is a choice in all marked columns. Returns boolean False
otherwise."""
        for column_index in self._marked_columns:
            if not self._choices[:, column_index].any():
                return False
        return True

    def __find_marked_column_without_choice(self):
        """Find a marked column that does not have a choice."""
        for column_index in self._marked_columns:
            if not self._choices[:, column_index].any():
                return column_index
```

```python
        raise HungarianError(
            "Could not find a column without a choice. Failed to cover matrix zeros. Algorithm
has failed.")

    def __find_row_without_choice(self, choice_column_index):
        """Find a row without a choice in it for the column indexed. If a row does not exist then
    return None."""
        row_indices, = np.where(self._zero_locations[:, choice_column_index])
        for row_index in row_indices:
            if not self._choices[row_index].any():
                return row_index

        # All rows have choices. Return None.
        return None

    def __find_best_choice_row_and_new_column(self, choice_column_index):
        """
        Find a row index to use for the choice so that the column that needs to be changed is
optimal.
        Return a random row and column if unable to find an optimal selection.
        """
        row_indices, = np.where(self._zero_locations[:, choice_column_index])
        for row_index in row_indices:
            column_indices, = np.where(self._choices[row_index])
            column_index = column_indices[0]
            if self.__find_row_without_choice(column_index) is not None:
                return row_index, column_index

        # Cannot find optimal row and column. Return a random row and column.
        from random import shuffle

        shuffle(row_indices)
        column_index, = np.where(self._choices[row_indices[0]])
        return row_indices[0], column_index[0]


if __name__ == '__main__':

    cost_matrix = [
        [4, 3, 6, 2, 7],
        [10, 12, 11, 14, 16],
        [4, 3, 2, 1, 5],
        [8, 7, 6, 9, 6]]

    hungarian = Hungarian(cost_matrix)
    hungarian.calculate()
    print("Cost Matrix : \n", np.array(cost_matrix))
    jobs = [1, 2, 3, 4]
    machine = ['A', 'B', 'C', 'D', 'E']
    result = hungarian.get_results()
    print("Job-Machine Assignment:")
    for i in result:
        print("Job-", jobs[i[0]], "-> Machine -", machine[i[1]], " is having Cost =", cost_matrix
[i[0]][i[1]])
    print("Total Processing Time:\t", hungarian.get_total_potential())
    print("-" * 80)
```

Output :


pythonProject1 – assignment5problem1.py
"/home/arghya/My Work/Python/pythonProject1/venv/bin/python" "/home/arghya/My Work/Python/pythonProject1/assignment5problem1.py"
Cost Matrix :
[[4 3 6 2 7]
[10 12 11 14 16]
[4 3 2 1 5]
[8 7 6 9 6]]
Job-Machine Assignment:
Job- 2 -> Machine - A is having Cost = 10
Job- 1 -> Machine - B is having Cost = 3
Job- 3 -> Machine - D is having Cost = 1
Job- 4 -> Machine - C is having Cost = 6
Total Processing Time: 20
Process finished with exit code 0

Problem 2

Problem statement :-

Four different jobs can be done on four different machines and the take-down time costs are prohibitively high for change overs. The matrix below gives the cost in rupees for producing job i on machine j.

		Machines		
Jobs	M_1	M_2	M_3	M_4
1	5	7	11	6
2	8	5	9	6
3	4	7	10	7
4	10	4	8	3

How the jobs should be assigned to the various machines so that the total cost is minimized.

Problem formulation :-

Checking for Balanced and unbalanced problem.

no of rows = 4 & no of col = 4
hence the problem is balanced

	M_1	M_2	M_3	M_4
1	x_{11} 5	x_{12} 7	x_{13} 11	x_{14} 6
2	x_{21} 8	x_{22} 5	x_{23} 9	x_{24} 6
3	x_{31} 4	x_{32} 7	x_{33} 10	x_{34} 7
4	x_{41} 10	x_{42} 4	x_{43} 8	x_{44} 3

objective $\min(z) = \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ij}$

$= 5x_{11} + 7x_{12} + 11x_{13} + 6x_{14} +$
$8x_{21} + 5x_{22} + 9x_{23} + 6x_{24} +$
$4x_{31} + 7x_{32} + 10x_{33} + 7x_{34} +$
$10x_{41} + 4x_{42} + 8x_{43} + 3x_{44}$

subject to constraint –

$x_{11} + x_{12} + x_{13} + x_{14} = 1 \rightarrow$ Job 1
$x_{21} + x_{22} + x_{23} + x_{24} = 1 \rightarrow$ Job 2
$x_{31} + x_{32} + x_{33} + x_{34} = 1 \rightarrow$ Job 3
$x_{41} + x_{42} + x_{43} + x_{44} = 1 \rightarrow$ Job 4

& $x_{11} + x_{21} + x_{31} + x_{41} \Rightarrow 1 \rightarrow$ Mach. 1
$x_{12} + x_{22} + x_{32} + x_{42} = 1 \rightarrow$ Mach 2
$x_{13} + x_{23} + x_{33} + x_{43} = 1 \rightarrow$ Mach 3
$x_{14} + x_{24} + x_{34} + x_{44} = 1 \rightarrow$ Mach 4.

and $x_{ij} \geq 0$ for $i, j = 1, 2, 3, 4$

Solutn

~~Remove zero by using~~

(1) Row & column reductn.

$$\begin{array}{c|cccc|c} & M_1 & M_2 & M_3 & M_4 & \\ \hline 1 & 5 & 7 & 11 & 6 & \rightarrow 5 \\ 2 & 8 & 5 & 9 & 6 & \rightarrow 5 \\ 3 & 4 & 7 & 10 & 7 & \rightarrow 4 \\ 4 & 10 & 4 & 8 & 3 & \rightarrow 3 \end{array}$$

$$\begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ \hline 1 & 0 & 2 & 6 & 1 \\ 2 & 3 & 0 & 4 & 1 \\ 3 & 0 & 3 & 6 & 3 \\ 4 & 7 & 1 & 5 & 0 \\ & \downarrow & \downarrow & \downarrow & \downarrow \\ & 0 & 0 & 4 & 0 \end{array}$$

$$\begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ \hline 1 & 0 & 2 & 2 & 1 \\ 2 & 3 & 0 & 0 & 1 \\ 3 & 0 & 3 & 2 & 3 \\ 4 & 7 & 1 & 1 & 0 \end{array}$$

(2) Remove zero by using min no of horizontal & vertical lines.

$$
\begin{array}{c|cccc}
 & M_1 & M_2 & M_3 & M_4 \\
\hline
1 & 0 & 2 & 2 & 1 \\
2 & 3 & 0 & 0 & 1 \\
3 & 0 & 3 & 2 & 3 \\
4 & 7 & 1 & 1 & 0
\end{array}
$$

no of lines = 3 < order of matrix = 4 hence choose min = 1 from uncovered values and substract it from non covered & add to intersection of lines and leave rest of them.

~~Step~~ 3

$$
\begin{array}{c|cccc}
 & M_1 & M_2 & M_3 & M_4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 4 & 0 & 0 & 1 \\
3 & 0 & 2 & 1 & 2 \\
4 & 8 & 1 & 1 & 0
\end{array}
$$

again no of lines 3 hence repeat the above process with min = 1

(4)

	M_1	M_2	M_3	M_4
1	0	0	0	0
2	5	0	0	2
3	0	1	0	2
4	8	0	0	0

so the no of lines = 4 == 4

hence now we go for '0' marking.

(5)

	M_1	M_2	M_3	M_4
1	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$
2	5	0	$\cancel{0}$	2
3	$\cancel{0}$	1	0	2
4	8	$\cancel{0}$	$\cancel{0}$	0

From the above table, we can get,

→ Job 1 is assigned to machine M_1 with cost = 5

→ Job 2 is assigned to mach. M_2 with cost = 5

→ Job 3 is assigned to M_3, cost = 10

→ Job 4 is assigned to M_4, cost = 3

Min total cost = 5 + 5 + 10 + 3
= 23

Python Code:

```
class HungarianError(Exception):
    pass

try:
    import numpy as np
except ImportError:
    raise HungarianError("NumPy is not installed.")


class Hungarian:
    """
    Implementation of the Hungarian (Munkres) Algorithm using np.
    Usage:
        hungarian = Hungarian(cost_matrix)
        hungarian.calculate()
    or
        hungarian = Hungarian()
        hungarian.calculate(cost_matrix)
    Handle Profit matrix:
        hungarian = Hungarian(profit_matrix, is_profit_matrix=True)
    or
        cost_matrix = Hungarian.make_cost_matrix(profit_matrix)
    The matrix will be automatically padded if it is not square.
    For that numpy's resize function is used, which automatically adds 0's to any row/column that
     is added
    Get results and total potential after calculation:
        hungarian.get_results()
        hungarian.get_total_potential()
    """

    def __init__(self, input_matrix=None, is_profit_matrix=False):
        """
        input_matrix is a List of Lists.
        input_matrix is assumed to be a cost matrix unless is_profit_matrix is True.
        """
        if input_matrix is not None:
            # Save input
            my_matrix = np.array(input_matrix)
            self._input_matrix = np.array(input_matrix)
            self._maxColumn = my_matrix.shape[1]
            self._maxRow = my_matrix.shape[0]

            # Adds 0s if any columns/rows are added. Otherwise stays unaltered
            matrix_size = max(self._maxColumn, self._maxRow)
            pad_columns = matrix_size - self._maxRow
            pad_rows = matrix_size - self._maxColumn
            my_matrix = np.pad(my_matrix, ((0, pad_columns), (0, pad_rows)), 'constant',
  constant_values=(0))

            # Convert matrix to profit matrix if necessary
            if is_profit_matrix:
                my_matrix = self.make_cost_matrix(my_matrix)

            self._cost_matrix = my_matrix
            self._size = len(my_matrix)
            self._shape = my_matrix.shape

            # Results from algorithm.
            self._results = []
            self._totalPotential = 0
        else:
            self._cost_matrix = None
```

```
    def get_results(self):
        """Get results after calculation."""
        return self._results

    def get_total_potential(self):
        """Returns expected value after calculation."""
        return self._totalPotential

    def calculate(self, input_matrix=None, is_profit_matrix=False):
        """
        Implementation of the Hungarian (Munkres) Algorithm.
        input_matrix is a List of Lists.
        input_matrix is assumed to be a cost matrix unless is_profit_matrix is True.
        """
        # Handle invalid and new matrix inputs.
        if input_matrix is None and self._cost_matrix is None:
            raise HungarianError("Invalid input")
        elif input_matrix is not None:
            self.__init__(input_matrix, is_profit_matrix)

        result_matrix = self._cost_matrix.copy()

        # Step 1: Subtract row mins from each row.
        for index, row in enumerate(result_matrix):
            result_matrix[index] -= row.min()

        # Step 2: Subtract column mins from each column.
        for index, column in enumerate(result_matrix.T):
            result_matrix[:, index] -= column.min()

        # Step 3: Use minimum number of lines to cover all zeros in the matrix.
        # If the total covered rows+columns is not equal to the matrix size then adjust matrix
and repeat.
        total_covered = 0
        while total_covered < self._size:
            # Find minimum number of lines to cover all zeros in the matrix and find total
covered rows and columns.
            cover_zeros = CoverZeros(result_matrix)
            covered_rows = cover_zeros.get_covered_rows()
            covered_columns = cover_zeros.get_covered_columns()
            total_covered = len(covered_rows) + len(covered_columns)

            # if the total covered rows+columns is not equal to the matrix size then adjust it by
 min uncovered num (m).
            if total_covered < self._size:
                result_matrix = self._adjust_matrix_by_min_uncovered_num(result_matrix,
covered_rows, covered_columns)

        # Step 4: Starting with the top row, work your way downwards as you make assignments.
        # Find single zeros in rows or columns.
        # Add them to final result and remove them and their associated row/column from the
matrix.
        expected_results = min(self._maxColumn, self._maxRow)
        zero_locations = (result_matrix == 0)
        while len(self._results) != expected_results:

            # If number of zeros in the matrix is zero before finding all the results then an
error has occurred.
            if not zero_locations.any():
                raise HungarianError("Unable to find results. Algorithm has failed.")

            # Find results and mark rows and columns for deletion
            matched_rows, matched_columns = self.__find_matches(zero_locations)

            # Make arbitrary selection
            total_matched = len(matched_rows) + len(matched_columns)
```

```
        if total_matched == 0:
            matched_rows, matched_columns = self.select_arbitrary_match(zero_locations)

        # Delete rows and columns
        for row in matched_rows:
            zero_locations[row] = False
        for column in matched_columns:
            zero_locations[:, column] = False

        # Save Results
        self.__set_results(zip(matched_rows, matched_columns))

    # Calculate total potential
    value = 0
    for row, column in self._results:
        value += self._input_matrix[row, column]
    self._totalPotential = value

@staticmethod
def make_cost_matrix(profit_matrix):
    """
    Converts a profit matrix into a cost matrix.
    Expects NumPy objects as input.
    """
    # subtract profit matrix from a matrix made of the max value of the profit matrix
    matrix_shape = profit_matrix.shape
    offset_matrix = np.ones(matrix_shape, dtype=int) * profit_matrix.max()
    cost_matrix = offset_matrix - profit_matrix
    return cost_matrix

def _adjust_matrix_by_min_uncovered_num(self, result_matrix, covered_rows, covered_columns):
    """Subtract m from every uncovered number and add m to every element covered with two
lines."""
    # Calculate minimum uncovered number (m)
    elements = []
    for row_index, row in enumerate(result_matrix):
        if row_index not in covered_rows:
            for index, element in enumerate(row):
                if index not in covered_columns:
                    elements.append(element)
    min_uncovered_num = min(elements)

    # Add m to every covered element
    adjusted_matrix = result_matrix
    for row in covered_rows:
        adjusted_matrix[row] += min_uncovered_num
    for column in covered_columns:
        adjusted_matrix[:, column] += min_uncovered_num

    # Subtract m from every element
    m_matrix = np.ones(self._shape, dtype=int) * min_uncovered_num
    adjusted_matrix -= m_matrix

    return adjusted_matrix

def __find_matches(self, zero_locations):
    """Returns rows and columns with matches in them."""
    marked_rows = np.array([], dtype=int)
    marked_columns = np.array([], dtype=int)

    # Mark rows and columns with matches
    # Iterate over rows
    for index, row in enumerate(zero_locations):
        row_index = np.array([index])
        if np.sum(row) == 1:
            column_index, = np.where(row)
```

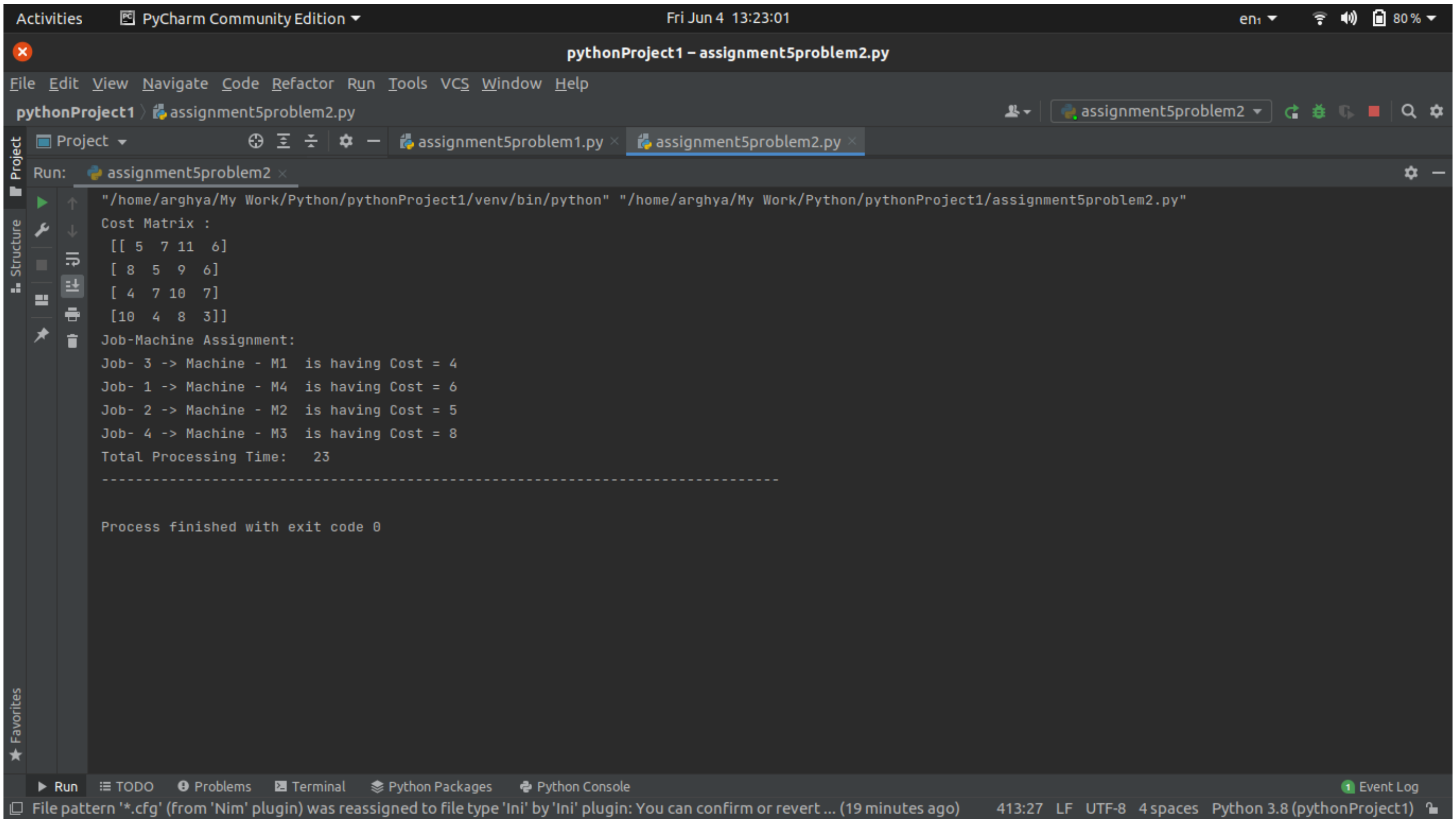
```
                marked_rows, marked_columns = self.__mark_rows_and_columns(marked_rows,
marked_columns, row_index,
                                                                            column_index)

        # Iterate over columns
        for index, column in enumerate(zero_locations.T):
            column_index = np.array([index])
            if np.sum(column) == 1:
                row_index, = np.where(column)
                marked_rows, marked_columns = self.__mark_rows_and_columns(marked_rows,
marked_columns, row_index,
                                                                            column_index)

        return marked_rows, marked_columns

    @staticmethod
    def __mark_rows_and_columns(marked_rows, marked_columns, row_index, column_index):
        """Check if column or row is marked. If not marked then mark it."""
        new_marked_rows = marked_rows
        new_marked_columns = marked_columns
        if not (marked_rows == row_index).any() and not (marked_columns == column_index).any():
            new_marked_rows = np.insert(marked_rows, len(marked_rows), row_index)
            new_marked_columns = np.insert(marked_columns, len(marked_columns), column_index)
        return new_marked_rows, new_marked_columns

    @staticmethod
    def select_arbitrary_match(zero_locations):
        """Selects row column combination with minimum number of zeros in it."""
        # Count number of zeros in row and column combinations
        rows, columns = np.where(zero_locations)
        zero_count = []
        for index, row in enumerate(rows):
            total_zeros = np.sum(zero_locations[row]) + np.sum(zero_locations[:, columns[index]])
            zero_count.append(total_zeros)

        # Get the row column combination with the minimum number of zeros.
        indices = zero_count.index(min(zero_count))
        row = np.array([rows[indices]])
        column = np.array([columns[indices]])

        return row, column

    def __set_results(self, result_lists):
        """Set results during calculation."""
        # Check if results values are out of bound from input matrix (because of matrix being
padded).
        # Add results to results list.
        for result in result_lists:
            row, column = result
            if row < self._maxRow and column < self._maxColumn:
                new_result = (int(row), int(column))
                self._results.append(new_result)


class CoverZeros:
    """
    Use minimum number of lines to cover all zeros in the matrix.
    Algorithm based on: http://weber.ucsd.edu/~vcrawfor/hungar.pdf
    """

    def __init__(self, matrix):
        """
        Input a matrix and save it as a boolean matrix to designate zero locations.
        Run calculation procedure to generate results.
        """
        # Find zeros in matrix
```

```python
        self._zero_locations = (matrix == 0)
        self._shape = matrix.shape

        # Choices starts without any choices made.
        self._choices = np.zeros(self._shape, dtype=bool)

        self._marked_rows = []
        self._marked_columns = []

        # marks rows and columns
        self.__calculate()

        # Draw lines through all unmarked rows and all marked columns.
        self._covered_rows = list(set(range(self._shape[0])) - set(self._marked_rows))
        self._covered_columns = self._marked_columns

    def get_covered_rows(self):
        """Return list of covered rows."""
        return self._covered_rows

    def get_covered_columns(self):
        """Return list of covered columns."""
        return self._covered_columns

    def __calculate(self):
        """
        Calculates minimum number of lines necessary to cover all zeros in a matrix.
        Algorithm based on: http://weber.ucsd.edu/~vcrawfor/hungar.pdf
        """
        while True:
            # Erase all marks.
            self._marked_rows = []
            self._marked_columns = []

            # Mark all rows in which no choice has been made.
            for index, row in enumerate(self._choices):
                if not row.any():
                    self._marked_rows.append(index)

            # If no marked rows then finish.
            if not self._marked_rows:
                return True

            # Mark all columns not already marked which have zeros in marked rows.
            num_marked_columns = self.__mark_new_columns_with_zeros_in_marked_rows()

            # If no new marked columns then finish.
            if num_marked_columns == 0:
                return True

            # While there is some choice in every marked column.
            while self.__choice_in_all_marked_columns():
                # Some Choice in every marked column.

                # Mark all rows not already marked which have choices in marked columns.
                num_marked_rows = self.__mark_new_rows_with_choices_in_marked_columns()

                # If no new marks then Finish.
                if num_marked_rows == 0:
                    return True

                # Mark all columns not already marked which have zeros in marked rows.
                num_marked_columns = self.__mark_new_columns_with_zeros_in_marked_rows()

                # If no new marked columns then finish.
                if num_marked_columns == 0:
```

```python
                return True

        # No choice in one or more marked columns.
        # Find a marked column that does not have a choice.
        choice_column_index = self.__find_marked_column_without_choice()

        while choice_column_index is not None:
            # Find a zero in the column indexed that does not have a row with a choice.
            choice_row_index = self.__find_row_without_choice(choice_column_index)

            # Check if an available row was found.
            new_choice_column_index = None
            if choice_row_index is None:
                # Find a good row to accomodate swap. Find its column pair.
                choice_row_index, new_choice_column_index = self.
__find_best_choice_row_and_new_column(
                    choice_column_index)

                # Delete old choice.
                self._choices[choice_row_index, new_choice_column_index] = False

            # Set zero to choice.
            self._choices[choice_row_index, choice_column_index] = True

            # Loop again if choice is added to a row with a choice already in it.
            choice_column_index = new_choice_column_index

 def __mark_new_columns_with_zeros_in_marked_rows(self):
     """Mark all columns not already marked which have zeros in marked rows."""
     num_marked_columns = 0
     for index, column in enumerate(self._zero_locations.T):
         if index not in self._marked_columns:
             if column.any():
                 row_indices, = np.where(column)
                 zeros_in_marked_rows = (set(self._marked_rows) & set(row_indices)) != set([])
                 if zeros_in_marked_rows:
                     self._marked_columns.append(index)
                     num_marked_columns += 1
     return num_marked_columns

 def __mark_new_rows_with_choices_in_marked_columns(self):
     """Mark all rows not already marked which have choices in marked columns."""
     num_marked_rows = 0
     for index, row in enumerate(self._choices):
         if index not in self._marked_rows:
             if row.any():
                 column_index, = np.where(row)
                 if column_index in self._marked_columns:
                     self._marked_rows.append(index)
                     num_marked_rows += 1
     return num_marked_rows

 def __choice_in_all_marked_columns(self):
     """Return Boolean True if there is a choice in all marked columns. Returns boolean False
otherwise."""
     for column_index in self._marked_columns:
         if not self._choices[:, column_index].any():
             return False
     return True

 def __find_marked_column_without_choice(self):
     """Find a marked column that does not have a choice."""
     for column_index in self._marked_columns:
         if not self._choices[:, column_index].any():
             return column_index
```

```python
            raise HungarianError(
                "Could not find a column without a choice. Failed to cover matrix zeros. Algorithm
    has failed.")

    def __find_row_without_choice(self, choice_column_index):
        """Find a row without a choice in it for the column indexed. If a row does not exist then
     return None."""
        row_indices, = np.where(self._zero_locations[:, choice_column_index])
        for row_index in row_indices:
            if not self._choices[row_index].any():
                return row_index

        # All rows have choices. Return None.
        return None

    def __find_best_choice_row_and_new_column(self, choice_column_index):
        """
        Find a row index to use for the choice so that the column that needs to be changed is
    optimal.
        Return a random row and column if unable to find an optimal selection.
        """
        row_indices, = np.where(self._zero_locations[:, choice_column_index])
        for row_index in row_indices:
            column_indices, = np.where(self._choices[row_index])
            column_index = column_indices[0]
            if self.__find_row_without_choice(column_index) is not None:
                return row_index, column_index

        # Cannot find optimal row and column. Return a random row and column.
        from random import shuffle

        shuffle(row_indices)
        column_index, = np.where(self._choices[row_indices[0]])
        return row_indices[0], column_index[0]


if __name__ == '__main__':
    cost_matrix = [
        [5, 7, 11, 6],
        [8, 5, 9, 6],
        [4, 7, 10, 7],
        [10, 4, 8, 3]]

    hungarian = Hungarian(cost_matrix)
    hungarian.calculate()
    print("Cost Matrix : \n", np.array(cost_matrix))
    jobs = [1, 2, 3, 4]
    machine = ['M1', 'M2', 'M3', 'M4']
    result = hungarian.get_results()
    print("Job-Machine Assignment:")
    for i in result:
        print("Job-", jobs[i[0]], "-> Machine -", machine[i[1]], " is having Cost =", cost_matrix
    [i[0]][i[1]])
    print("Total Processing Time:\t", hungarian.get_total_potential())
    print("-" * 80)
```

Output:


pythonProject1 – assignment5problem2.py
"/home/arghya/My Work/Python/pythonProject1/venv/bin/python" "/home/arghya/My Work/Python/pythonProject1/assignment5problem2.py"
Cost Matrix :
[[5 7 11 6]
[8 5 9 6]
[4 7 10 7]
[10 4 8 3]]
Job-Machine Assignment:
Job- 3 -> Machine - M1 is having Cost = 4
Job- 1 -> Machine - M4 is having Cost = 6
Job- 2 -> Machine - M2 is having Cost = 5
Job- 4 -> Machine - M3 is having Cost = 8
Total Processing Time: 23

Process finished with exit code 0