Assignment 4
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 1

Problem statement 1

Find the optimal solu^n to the following transportat^n problem in which the cells contains the unit transportat^n cost in ruppees.

	w_1	w_2	w_3	w_4	w_5	avail.
F_1	7	6	4	5	9	40
F_2	8	5	6	7	8	30
F_3	6	8	9	6	5	20
F_4	5	7	7	8	6	10
Req.	30	30	15	20	5	

use NWCR and LCM for initial basic feasible solu^n.

Problem formulat^n –

	W_1	W_2	W_3	W_4	W_5	Supply
F_1	x_{11} 7	x_{12} 6	x_{13} 4	x_{14} 5	x_{15} 9	40
F_2	x_{21} 8	x_{22} 5	x_{23} 6	x_{24} 7	x_{25} 8	30
F_3	x_{31} 6	x_{32} 8	x_{33} 9	x_{34} 6	x_{35} 5	20
~~Demand~~ F_4	x_{41} 5	x_{42} 7	x_{43} 7	x_{44} 8	x_{45} 6	10
Dem.	30	30	15	20	5	

The transportatn problem is formulated as an LP model as follows —

Minimize (Total T.P cost) —

$$7x_{11} + 6x_{12} + 4x_{13} + 5x_{14} + 9x_{15} \cancel{\le 40}$$
$$+ 8x_{21} + 5x_{22} + 6x_{23} + 7x_{24} + 8x_{25} \cancel{\le 30}$$
$$+ 6x_{31} + 8x_{32} + 9x_{33} + 6x_{34} + 5x_{35} \cancel{\le 20}$$
$$+ 5x_{41} + 7x_{42} + 7x_{43} + 8x_{44} + 6x_{45} \cancel{\le 10}$$

Subject to constraint —

$$7x_{11} + 6x_{12} + 4x_{13} + 5x_{14} + 9x_{15} = 40$$
$$8x_{21} + 5x_{22} + 6x_{23} + 7x_{24} + 8x_{25} = 30$$
$$6x_{31} + 8x_{32} + 9x_{33} + 6x_{34} + 5x_{35} = 20$$
$$5x_{41} + 7x_{42} + 7x_{43} + 8x_{44} + 6x_{45} = 10$$

First we will solve it using north west corner rule.

The allocatⁿ matrix is such that.

	W_1	W_2	W_3	W_4	W_5	Sup
F_1	7 (30)	6 (10)	4	5	9	~~40~~ ~~10~~ 0
F_2	8	5 (20)	6 (10)	7	8	~~30~~ ~~10~~ 0
F_3	6	8	9 (5)	6 (15)	5	~~20~~ ~~15~~ 0
F_4	5	7	7	8 (5)	6	10 ~~5~~ 0
Demand	~~30~~ 0	~~30~~ ~~20~~ 0	~~15~~ ~~5~~ 0	~~20~~ ~~15~~ ~~5~~ 0	~~5~~ 0	

The step by step descriptⁿ of this solutⁿ is given as:-

and

$$7x_{11} + 8x_{21} + 6x_{31} + 5x_{41} = 30$$

$$6x_{12} + 5x_{22} + 8x_{32} + 7x_{42} = 30$$

$$4x_{13} + 6x_{23} + 9x_{33} + 7x_{43} = 15$$

$$5x_{14} + 7x_{24} + 6x_{34} + 8x_{44} = 20$$

$$9x_{15} + 8x_{25} + 5x_{35} + 6x_{45} = 5$$

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

Soln

Total no of supply const. = 4

Total no of demand const = 5

~~The sym val for $t_1 = 40$ & $a_2 = 30$ compared.~~

Total supply = $40 + 30 + 20 + 10$
$= 100$

Total demand = $30 + 30 + 15 + 20 + 5$
$= 100$

Total supply = Total demand

$\Rightarrow$ Balanced Transp. problem.

North west corner method

The sim val $F_1 = 40$ & $W_1 = 30$ $\min(F_1, W_1) = 30$ is assigned to $F_1 W_1$, This meets the demand of W_1 & leaves $40 - 30 = 10$ with F_1

The sim value for $F_2 = 30$ & $W_2 = 20$ are compared. The $\min(F_2, W_2) = 20$. This meets the demand of W_2 & leaves $30 - 20 = 10$ units with F_2

The sim values for $F_2 = 10$ and $W_3 = 15$ are compared. The smaller of the two i.e $\min(10, 15) = 10$ is assign. to $F_2 W_3$, this exhaust

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/c
    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'

            # 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
        total_covered = 0
        while total_covered < self._size:
            # Find minimum number of lines to cover all zeros in the matrix and find to
            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 ad
            if total_covered < self._size:
                result_matrix = self._adjust_matrix_by_min_uncovered_num(result_matrix,
```

```
        # Step 4: Starting with the top row, work your way downwards as you make assignm
        # Find single zeros in rows or columns.
        # Add them to final result and remove them and their associated row/column from
        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 t
            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_locati

            # 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 matr
        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_c
        """Subtract m from every uncovered number and add m to every element covered wi
        # 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,
                                                                        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,
                                                                        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
        new_marked_rows = np.insert(marked_rows, len(marked_rows), row_index)
        new_marked_columns = np.insert(marked_columns, len(marked_columns), column_
    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[:, column
            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
        # 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
        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."""
```

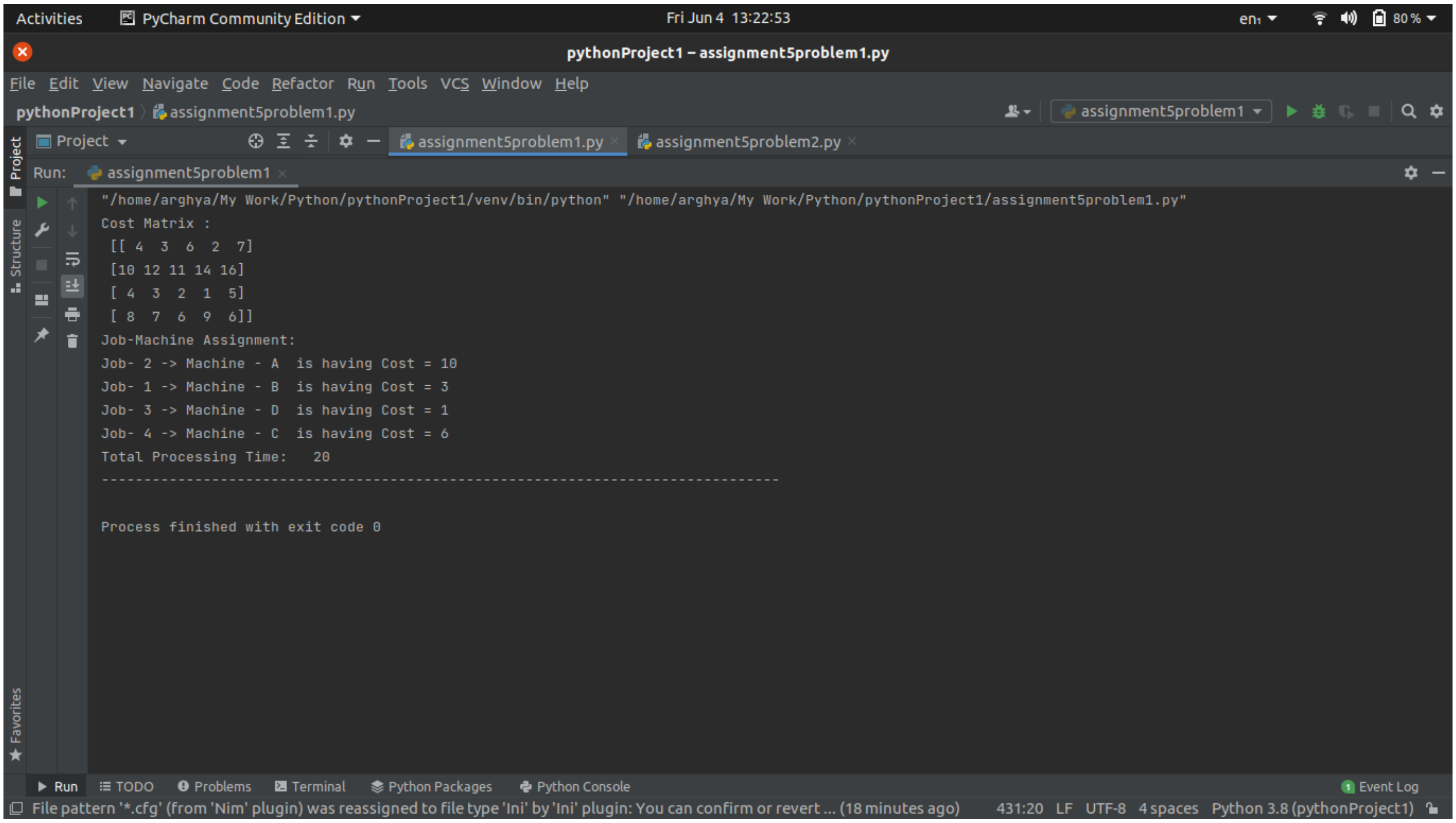
```
        return self._covered_rows

    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 cho
            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_r
                    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))
                    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 boole
        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

        raise HungarianError(
            "Could not find a column without a choice. Failed to cover matrix zeros. Al

    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
        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 chan
        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 =", c
```

```
print("Total Processing Time:\t", hungarian.get_total_potential())
print("-" * 80)
```

Output :


Activities
PyCharm Community Edition
Fri Jun 4 13:22:53
pythonProject1 – assignment5problem1.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject1 assignment5problem1.py
assignment5problem1
Project
assignment5problem1.py
assignment5problem2.py
Run: assignment5problem1
"/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
Run TODO Problems Terminal Python Packages Python Console
Event Log
File pattern '*.cfg' (from 'Nim' plugin) was reassigned to file type 'Ini' by 'Ini' plugin: You can confirm or revert ... (18 minutes ago)
431:20 LF UTF-8 4 spaces Python 3.8 (pythonProject1)

Problem 2

Problem statement 1

Find the optimal solutn to the following transportatn problem in which the cells contains the unit transportatn cost in ruppees.

	W_1	W_2	W_3	W_4	W_5	avail.
F_1	7	6	4	5	9	40
F_2	8	5	6	7	8	30
F_3	6	8	9	6	5	20
F_4	5	7	7	8	6	10
Req.	30	30	15	20	5	

use NWCR and LCM for initial basic feasible solutn.

Problem formulatn –

the capacity of F_2 and leaves 15 $-10 = 5$ units with W_3.

The min value for $F_3 = 20$ and $W_3 = 5$ are compared. The min $(20, 5) = 5$ is assigned to $F_3 W_3$ This meets the complete demand of W_3 and leaves $20 - 5 = 15$ units with F_3

The min values for $F_3 = 15$ and $W_4 = 20$ are compared.
The smaller of $15, 20 = 15$ is assigned for $F_3 W_4$, this exhaust the capacity of F_3 and leaves $20 - 15 = 5$ with W_4.

The min value for $F_4 = 10$ & $W_4 = 5$ are compared. The

min (10, 5) = 5 is assign to $F_4 W_4$
This meets the complete demand of W_4 and leaves 10 - 5 = 5 units with F_4

The min value for $F_4 = 5$ & $W_5 = 5$ are compared.
The smaller of the two i.e min (5, 5) = 5) is assign to $F_4 W_5$
The IBFS =

	W_1	W_2	W_3	W_4	W_5	supp.
F_1	7 (30)	6 (10)	4	5	9	40
F_2	8	5 (20)	6 (10)	7	8	30
F_3	6	8	9 (5)	6 (15)	5	20
F_4	5	7	7	8 (5)	6 (5)	10
Demand	30	30	15	20	5	

$v_1 = 7 - 0 = 7$

$v_2 = 6 - Q =$

$v_1 = c_{11} - u_1 = 7 - 0 \Rightarrow v_1 = 7$

$v_2 = c_{12} - u_1 = 6 - 0 \Rightarrow v_2 = 6$

$u_2 = c_{22} - v_2 = 5 - 6 \Rightarrow u_2 = -1$

$v_3 = c_{23} - u_2 = 6 + 1 \Rightarrow v_3 = 7$

$u_3 = c_{33} - v_3 = 9 - 7 \Rightarrow u_3 = 2$

$u_4 = c_{44} - v_4 = 8 - 4 \Rightarrow u_4 = 4$

$v_5 = c_{45} - u_4 = 6 - 4 = v_5 = 2$

	W_1	W_2	W_3	W_4	W_5	S	u_i
F_1	7 (30)	6 (10)	4	5	9	40	$u_1 = 0$
F_2	8	5 (20)	6 (10)	7	8	30	$u_2 = -1$
F_3	6	8	9 (5)	6 (15)	5	20	$u_3 = 2$
	5	7	7	8 (5)	6 (5)	10	$u_4 = 4$
	30	30	15	20	5		
v_j	$v_1 = 7$	$v_2 = 6$	$v_3 = 7$	$v_4 = 4$	$v_5 = 2$		

The minimization T.P. cost =

$7 \times 30 + 6 \times 10 + 5 \times 20 + 6 \times 10 + 9 \times 5$
$+ 6 \times 15 + 8 \times 5 + 6 \times 5 = 635$

Total allocated cell = 8 = m+n-1
= 4+5-1
= 8

hence the solⁿ is non degenerate

optimality test using modi method.

Allocation table is

	W_1	W_2	W_3	W_4	W_5	Supp.
F_1	7 (30)	6 (10)	9	5	9	40
F_2	8	5 (20)	6 (10)	7	8	30
F_3	6	8	9 (5)	6 (15)	5	20
F_4	5	7	7	8 (5)	6 (5)	10
Dem.	30	30	15	20	5	

Iteratⁿ 1 of optimality test.

(1) Subs. $u_1 = 0$, we get

(2) find $d_{ij} = c_{ij} - (u_i + v_j)$

$d_{13} = c_{13} - (u_1 + v_3) = 4 - (0+7) = -3$

$d_{14} = 5 - (0+4) = 1$

$d_{15} = 9 - (0+2) = 7$

$d_{21} = 8 - (-1+7) = 2$

$d_{24} = 7 - (-1+4) = 4$

$d_{25} = 8 - (-1+2) = 7$

$d_{31} = 6 - (2+7) = -3$

$d_{32} = 8 - (2+6) = 0$

$d_{35} = 5 - (2+2) = 1$

$d_{41} = 5 - (4+7) = -6$

$d_{42} = 7 - (4+6) = -3$

$d_{43} = 7 - (4+7) = -4$

3)

	w_1	w_2	w_3	w_4	w_5	supp	u_i
t_1	7 (30) (−)	6 (10) (+)	4^{-3}	5^{1}	9^{7}	40	0
t_2	8^{2}	5 (20) (−)	6 (10) (+)	7^{4}	8^{7}	30	−1
t_3	6^{-3}	8^{0}	9 (5) (−)	6 (15) (+)	5^{1}	20	2
t_4	5^{-6} (+)	7^{-3}	7^{-4}	8 (5) (−)	6 (5)	10	4
dem	30	30	15	20	5		
v_j	7	6	7	4	2		

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/co
    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'

            # 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
        total_covered = 0
        while total_covered < self._size:
            # Find minimum number of lines to cover all zeros in the matrix and find to
            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 ad
            if total_covered < self._size:
                result_matrix = self._adjust_matrix_by_min_uncovered_num(result_matrix,
```

```
        # Step 4: Starting with the top row, work your way downwards as you make assignn
        # Find single zeros in rows or columns.
        # Add them to final result and remove them and their associated row/column from
        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 tl
            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_locatic

            # 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 matr
        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_c
        """Subtract m from every uncovered number and add m to every element covered wi
        # 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,
                                                                       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,
                                                                       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
        new_marked_rows = np.insert(marked_rows, len(marked_rows), row_index)
        new_marked_columns = np.insert(marked_columns, len(marked_columns), column_
    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[:, column
            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
        # 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
        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."""
```

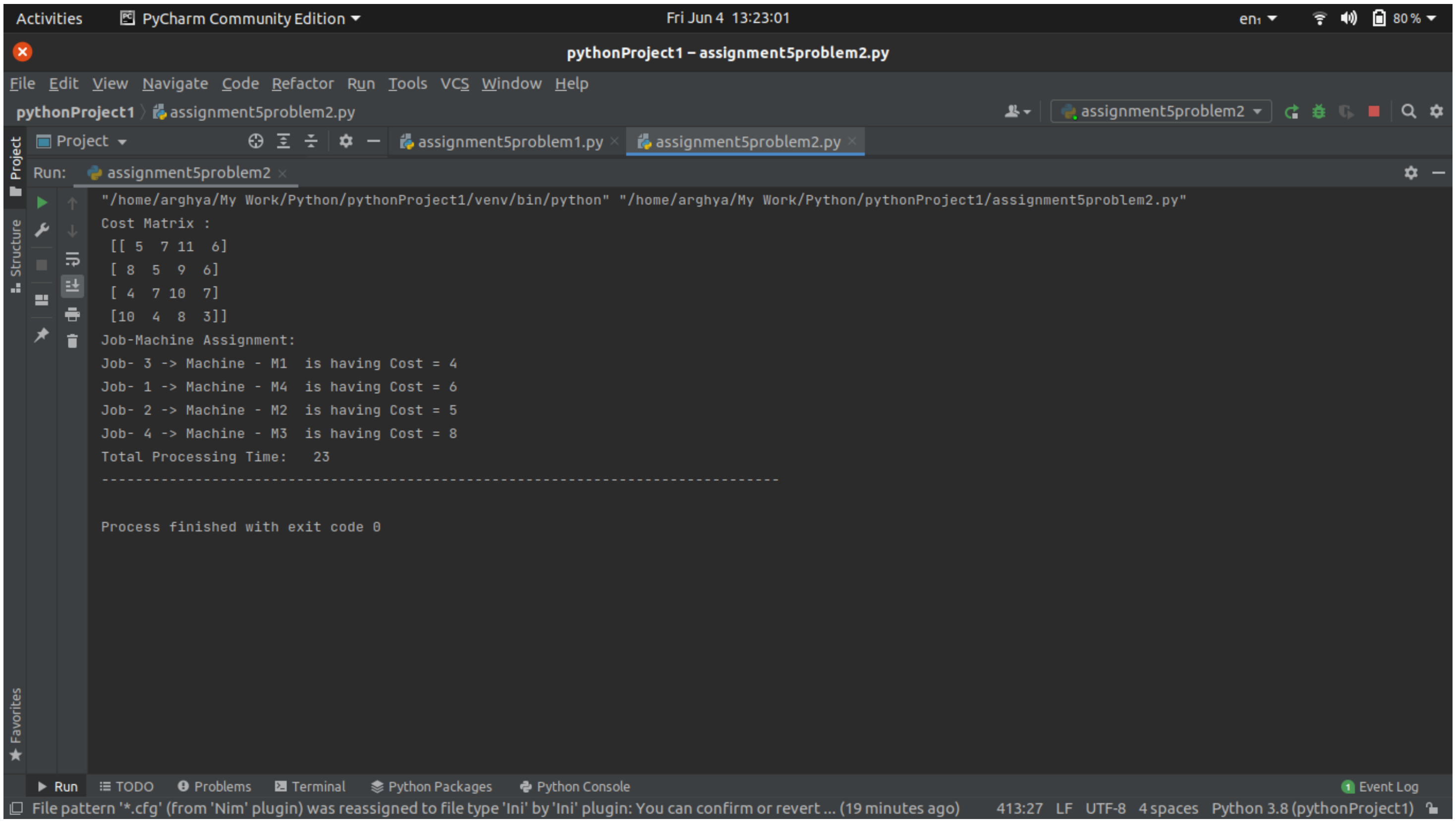
```
        return self._covered_rows

    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 cho
            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_r
                    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))
                    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 boole
        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

        raise HungarianError(
            "Could not find a column without a choice. Failed to cover matrix zeros. Al

    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
        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 chan
        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 =", c
    print("Total Processing Time:\t", hungarian.get_total_potential())
```

```
print("-" * 80)
```

Output :


Activities
PyCharm Community Edition
Fri Jun 4 13:23:01
80 %
pythonProject1 – assignment5problem2.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject1 assignment5problem2.py
assignment5problem2
Project
assignment5problem1.py
assignment5problem2.py
Run: assignment5problem2
"/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
Run
TODO
Problems
Terminal
Python Packages
Python Console
Event Log
File pattern '*.cfg' (from 'Nim' plugin) was reassigned to file type 'Ini' by 'Ini' plugin: You can confirm or revert ... (19 minutes ago)
413:27 LF UTF-8 4 spaces Python 3.8 (pythonProject1)
Project
Structure
Favorites