Report 3

Team information.

- Group: ISE-02
- Team leader: Arthur Popov
- Team member 1: Arthur Popov — *grade - 5*
- Team member 2: Ivan Murzin — *grade - 5*
- Team member 3: Egor Glebov — *grade - 5*
- Team member 4: Adelina Karavaeva — *grade - 5*

Link to the product.

- The product is available: `https://github.com/pop-arthur/OptimizationCollab.git`

Programming language.

- Programming language: *Python*

Input

The input contains:

- A vector of coefficients of supply - S.
- A matrix of coefficients of costs - C.
- A vector of coefficients of demand - D.

Output/Results

The output contains one of the following:

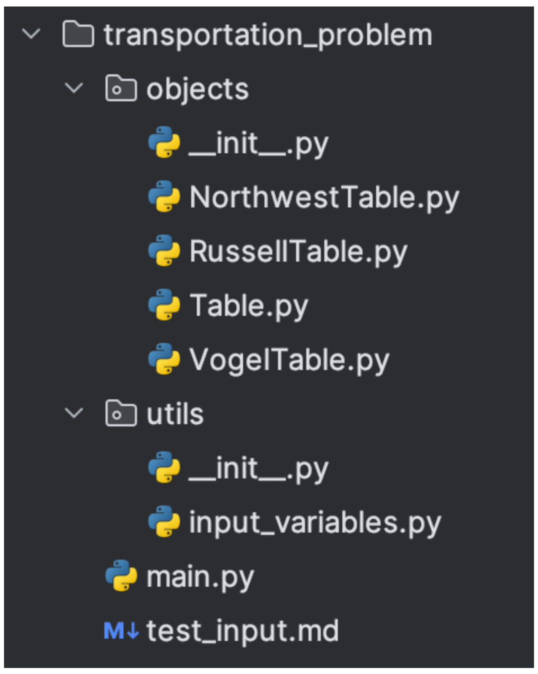
- The string The method is not applicable!
- The string The problem is not balanced!
- Print (demonstrate) input parameter table (a table constructed using matrix C, vectors S and D), 3 vectors of initial basic feasible solution - x^0 using North-West corner method, Vogels approximation method, and Russells approximation method.

Tests

Five tests are shown in file ”test_input.md” that presented on pages 3-4.

Code

We have file system structure shown in Fig. 1 :



```
transportation_problem
    objects
        __init__.py
        NorthwestTable.py
        RussellTable.py
        Table.py
        VogelTable.py
    utils
        __init__.py
        input_variables.py
    main.py
    test_input.md
```

Figure 1: files organization

You can observe full code on github or follow code snippets below.

main.py

```
from utils import *
from objects import *


def main():
    # input data
    supply, cost, demand = input_variables()

    # create transportation table
    table = Table(supply, cost, demand)
    # show the table
    table.show()
    # print results
    print(
        "",
        f"North-West Corner method:", NorthwestTable(table).get_solution(),
        f"Vogel s approximation method:", VogelTable(table).get_solution(),
        f"Russell s approximation method:", RussellTable(table).get_solution
                                            (),
        sep="\n"
    )


if __name__ == '__main__':
    # except possible errors
    try:
        # call main function
        main()
    except ValueError as e:
        print(e)
```

test_input.md

~~~
# Applicable

## Problem 1
Lab 7 problem 1 (NW)
```
140 180 160
2 3 4 2
8 4 1 4
9 7 3 7
100 120 160 100
```
Answer:
```
          B1    B2    B3    B4    Supply
------  ----  ----  ----  ----  --------
A1         2     3     4     2       140
A2         8     4     1     4       180
A3         9     7     3     7       160
Demand   100   120   160   100       480

North-West Corner method:
['100 * A1B1 (2)', '40 * A1B2 (3)', '80 * A2B2 (4)', '100 * A2B3 (1)', '60 *
                                        A3B3 (3)', '100 * A3B4 (7)']
 Vogel s approximation method:
['100 * A1B1 (2)', '160 * A3B3 (3)', '40 * A1B4 (2)', '120 * A2B2 (4)', '60 *
                                        A2B4 (4)']
 Russell s approximation method:
['100 * A1B1 (2)', '40 * A1B4 (2)', '120 * A2B2 (4)', '60 * A2B4 (4)', '160 *
                                        A3B3 (3)']
```

## Problem 2
Lab 7 problem 3 (Vogel)
```
160 140 170
7 8 1 2
4 5 9 8
9 2 3 6
120 50 190 110
```
Answer:
```
          B1    B2    B3    B4    Supply
------  ----  ----  ----  ----  --------
A1         7     8     1     2       160
A2         4     5     9     8       140
A3         9     2     3     6       170
Demand   120    50   190   110       470

North-West Corner method:
['120 * A1B1 (7)', '40 * A1B2 (8)', '10 * A2B2 (5)', '130 * A2B3 (9)', '60 *
                                        A3B3 (3)', '110 * A3B4 (6)']
 Vogel s approximation method:
['110 * A1B4 (2)', '50 * A1B3 (1)', '140 * A3B3 (3)', '30 * A3B2 (2)', '120 *
                                        A2B1 (4)', '20 * A2B2 (5)']
 Russell s approximation method:
['160 * A1B3 (1)', '30 * A3B3 (3)', '120 * A2B1 (4)', '50 * A3B2 (2)', '20 *
                                        A2B4 (8)', '90 * A3B4 (6)']
```
~~~

````
## Problem 3
[Link](https://cbom.atozmath.com/example/CBOM/Transportation.aspx?q=ram) (
                                        Russel)
‘‘‘
7   9   18
19  30  50  10
70  30  40  60
40  8 70  20
5 8 7 14
‘‘‘
Answer:
‘‘‘
          B1    B2    B3    B4    Supply
------  ----  ----  ----  ----  --------
A1        19    30    50    10         7
A2        70    30    40    60         9
A3        40     8    70    20        18
Demand     5     8     7    14        34

North-West Corner method:
['5 * A1B1 (19)', '2 * A1B2 (30)', '6 * A2B2 (30)', '3 * A2B3 (40)', '4 * A3B3
                                        (70)', '14 * A3B4 (20)']
 V o g e l  s approximation method:
['8 * A3B2 (8)', '5 * A1B1 (19)', '10 * A3B4 (20)', '2 * A1B4 (10)', '7 * A2B3
                                        (40)', '2 * A2B4 (60)']
 R u s s e l l  s approximation method:
['14 * A3B4 (20)', '5 * A1B1 (19)', '4 * A3B2 (8)', '2 * A1B2 (30)', '2 * A2B2
                                        (30)', '7 * A2B3 (40)']
‘‘‘

# Not balanced
‘‘‘
140 180 150
2 3 4 2
8 4 1 4
9 7 3 7
70 120 130 100
‘‘‘
Answer: ‘The problem is not balanced!‘

# Not applicable
‘‘‘
7   9   18
19  30  50  10
70  30  40
40  8 70  20
5 8 7 14
‘‘‘
Answer: ‘The method is not applicable!‘
````

objects/Table.py

```
from tabulate import tabulate

# class representing transportation table
class Table:
    def __init__(self, supply, costs, demand):
        """
        :param supply: vector of supply
        :param costs: matrix of costs
        :param demand: vector of demand
        """
        self.table = [
            [*costs[i], supply[i]] for i in range(len(supply))
        ] + [[*demand, sum(demand)]]

        self.supply = supply
        self.costs = costs
        self.demand = demand

    def __str__(self):
        return str(self.table)

    def show(self):
        print(
            tabulate(
                self.table,
                headers=[*[f"B{i + 1}" for i in range(len(self.table[0]) - 1)],
                                                "Supply"],
                showindex=[f"A{i + 1}" for i in range(len(self.table) - 1)] + [
                                               "Demand"],
            )
        )

    # abstract method that should be defined in subclasses
    def get_solution(self):
        pass

    def process_subtraction(self, x, y):
        """
        method that process operations over table when solution is defined
        :param x: x coordinate of solution
        :param y: y coordinate of solution
        :return: solution string
        """
        # get possible amount
        amount = min(self.supply[x], self.demand[y])
        # get price of transportation
        price = self.costs[x][y]
        # add cell to the solution vector
        # subtract cell, supply and demand
        self.costs[x][y] = 0
        self.supply[x] -= amount
        self.demand[y] -= amount
        # process null supply
        if self.supply[x] == 0:
            for j in range(len(self.costs[x])):
                self.costs[x][j] = 0
        # process null demand
        if self.demand[y] == 0:
            for i in range(len(self.costs)):
                self.costs[i][y] = 0
        return f"{amount} * A{x + 1}B{y + 1} ({price})"
```

objects/NorthwestTable.py

```
from transportation_problem.objects.Table import Table
from copy import deepcopy


# class for North-West corner method
class NorthwestTable(Table):
    def __init__(self, table: Table):
        super().__init__(table.supply.copy(), deepcopy(table.costs), table.
                                                       demand.copy())

    def find_non_empty_northwest_cell(self):
        """
        method finds non-empty northwest cell in costs list
        :return: coordinates of non-empty NW cell
        """
        for i in range(len(self.costs)):
            for j in range(len(self.costs[i])):
                if self.costs[i][j] != 0:
                    return i, j
        return None, None

    def get_solution(self):
        """
        method for North-West corner method
        :return: vector of solution x0
        """
        # vector of solution
        solution = []
        while True:
            # get northwest non-empty cell
            x, y = self.find_non_empty_northwest_cell()
            # end if cell is not found
            if (x, y) == (None, None):
                break

            # subtract demand and supply and add solution
            solution.append(self.process_subtraction(x, y))

        # return found solution
        return solution
```

objects/RussellTable.py

```
from transportation_problem.objects.Table import Table
from copy import deepcopy


# class for Russell's approximation method
class RussellTable(Table):
    def __init__(self, table: Table):
        super().__init__(table.supply.copy(), deepcopy(table.costs), table.
                                                       demand.copy())

    def get_max_in_the_row(self):
        """
        :return: list of maximum values in each row
        """
        return [max(row) for row in self.costs]

    def get_max_in_the_column(self):
        """
        :return: list of maximum values in each column
        """
        res = []
        for j in range(len(self.costs[0])):
            res.append(
                max(self.costs[i][j] for i in range(len(self.costs)))
            )
        return res

    def get_solution(self):
        """
        method for Russell's approximation method
        :return: vector of solution x0
        """
        # vector of solution
        solution = []
        while True:
            # get vectors of maximum rows / columns
            row_max, col_max = self.get_max_in_the_row(), self.
                                                       get_max_in_the_column()
            # if everything is done, then end
            if max(row_max) == 0 and max(col_max) == 0:
                break
            # copy array of costs
            costs = deepcopy(self.costs)
            # calculate cij
            for i in range(len(costs)):
                for j in range(len(costs[0])):
                    if costs[i][j] != 0:
                        costs[i][j] = costs[i][j] - (row_max[i] + col_max[j])

            # choose the most negative value in the table
            x, y = None, None
            value = 10 ** 3
            for i in range(len(costs)):
                for j in range(len(costs[0])):
                    if costs[i][j] < value:
                        x = i
                        y = j
                        value = costs[i][j]
            if value >= 0:
                break
            # subtract demand and supply and add solution
            solution.append(self.process_subtraction(x, y))

        return solution
```

objects/VogelTable.py

```python
from transportation_problem.objects.Table import Table
from copy import deepcopy


# class for Vogel's approximation method
class VogelTable(Table):
    def __init__(self, table: Table):
        super().__init__(table.supply.copy(), deepcopy(table.costs), table.
                                                        demand.copy())

    def get_rows_delta(self):
        """
        :return: list of differences between two minimum numbers in each row
        """
        delta_rows = []
        for i in range(len(self.costs)):
            sorted_row = list(filter(lambda x: x != 0, sorted(self.costs[i])))
            if len(sorted_row) >= 2:
                delta_rows.append(sorted_row[1] - sorted_row[0])
            elif len(sorted_row) == 1:
                delta_rows.append(1)
            else:
                delta_rows.append(0)

        return delta_rows

    def get_columns_delta(self):
        """
        :return: list of differences between two minimum numbers in each column
        """
        delta_columns = []
        for j in range(len(self.costs[0])):
            sorted_column = sorted([self.costs[i][j] for i in range(len(self.
                                                        costs)) if self.costs[i][j]
                                                         != 0])
            if len(sorted_column) >= 2:
                delta_columns.append(sorted_column[1] - sorted_column[0])
            elif len(sorted_column) == 1:
                delta_columns.append(1)
            else:
                delta_columns.append(0)
        return delta_columns

    def get_solution(self):
        """
        method for Vogel's approximation method
        :return: vector of solution x0
        """
        # vector of solution
        solution = []
        while True:
            # get maximum difference between two min numbers
            rows_delta = self.get_rows_delta()
            max_delta_row = max(rows_delta)
            columns_delta = self.get_columns_delta()
            max_delta_column = max(columns_delta)

            # if end reached
            if max_delta_row == max_delta_column == 0:
                break
            # if maximum difference in row
            elif max_delta_row >= max_delta_column:
                # get number of row with maximum difference
                row_number = rows_delta.index(max_delta_row)
                # get coordinates of minimum cell in the row
                x, y = row_number, self.costs[row_number].index(
                    min(list(filter(lambda t: t != 0, self.costs[row_number])))
                )
            # if maximum difference in column
            else:
                # get number of column with maximum difference
                column_number = columns_delta.index(max_delta_column)
                # column numbers
                column_numbers = [self.costs[i][column_number] for i in range(
                                                        len(self.costs))]
                # get coordinates of cell
                x, y = column_numbers.index(
                    min(list(filter(lambda t: t != 0, column_numbers)))
                ), column_number

            # subtract demand and supply and add solution
            solution.append(self.process_subtraction(x, y))

        return solution
```

utils/input_variables.py

```python
def input_variables():
    supply = list(map(int, input().split()))
    costs = [list(map(int, input().split())) for _ in range(len(supply))]
    demand = list(map(int, input().split()))
    if len(supply) != len(costs) or any(len(demand) != len(costs[i]) for i in
                                            range(len(costs))):
        raise ValueError('The method is not applicable!')
    if sum(supply) != sum(demand):
        raise ValueError("The problem is not balanced!")
    return supply, costs, demand
```