Name: Girish S Roll No.: AM.EN.U4AIE22044

-22AIE203-
DATA STRUCTURES AND ALGORITHMS -2

ASSIGNMENT 5 – Sparse Matrix and Block Matrix

Sparse Matrix

```
class SparseMatrix:

    def __init__(self, rows, cols):
        self.__data = [[rows, cols, 0]]

    def __setitem__(self, key, value):
        if (key[0] > self.__data[0][0]-1) or (key[1] > self.__data[0][1]-1):
            print(f"Invalid Indexing: {key} for size of
{tuple(self.__data[0][:2])}")
        if self[key] == 0:
            self.__data[0][2]+=1
            self.__data.append([key[0], key[1], value])
            header = self.__data[0]
            data = sorted(self.__data[1:])
            self.__data = [header]
            for i in data:
                self.__data.append(i)
        else:
            header = self.__data[0]
            self.__data = [[i[0], i[1], value] if tuple(i[:2]) == key else
i for i in self.__data[1:]]
            self.__data.insert(0, header)


    def __getitem__(self, key):
        if (key[0] > self.__data[0][0]-1) or (key[1] > self.__data[0][1]-1):
            print(f"Invalid Indexing: {key} for size of
{(self.__data[0][0]-1, self.__data[0][1]-1)}")
            return None
```

```
            for i in self.__data[1:]:
                if (i[0], i[1]) == key:
                    return i[2]
            return 0

        def __add__(self, other):
            if self.__data[0][:2] == other.__data[0][:2]:
                header = self.__data[0]
                newinstance = SparseMatrix(header[0], header[1])

                for i in self.__data[1:]:
                    newinstance[(i[0], i[1])] = i[2]
                for i in other.__data[1:]:
                    newinstance[(i[0], i[1])] = newinstance[(i[0], i[1])] +
i[2]
                return newinstance
            else:
                print(f"Dimension Error: the Dimensions ({self.__data[0][0]},
{self.__data[0][1]}) and ({other.__data[0][0]}, {other.__data[0][1]}) doesn't
match")

        def __str__(self):
            res=""
            data = [[0 for i in range(self.__data[0][1])] for i in
range(self.__data[0][0])]          #[[0]*self.__data[0][1]]*self.__data[0][0]
            for i in self.__data[1:]:
                data[i[0]][i[1]] = i[2]
            for i in data:
                for j in i:
                    res+="{:>5}".format(round(j, 2))
                res+="\n"
            return res[:-1]

        def display(self):
            print("+-----+-----+-----+-----+")
            print("| Ind | Row | Col | Val |")
            print("+-----+-----+-----+-----+")
            for i, val in enumerate(self.__data):
                if i==0:
                    print("| {:<3} | {:<3} | {:<3} | {:<3} |".format(i+1,
val[0], val[1], val[2]))
                    print("+-----+-----+-----+-----+")
                    continue
                print("| {:<3} | {:<3} | {:<3} | {:<3} |".format(i+1, val[0]+1,
val[1]+1, val[2]))
            print("+-----+-----+-----+-----+")

        def transpose(self):
            for i in self.__data:
                i[0], i[1] = i[1], i[0]
```

```
    def inverse(self):
        if self.__data[0][0] != self.__data[0][1]:
            print("Inverse Error: The Matrix is not a square matrix")
            return
        if self.determinant() == 0:
            print("Inverse Error: The Matrix is Indefinite Matrix")
            return
        matrix = [[0 for _ in range(self.__data[0][1])] for _ in
range(self.__data[0][0])]
        for i in self.__data[1:]:
            matrix[i[0]][i[1]] = i[2]
        n = len(matrix)
        identity = [[0] * n for _ in range(n)]
        for i in range(n):
           identity[i][i] = 1

        # Gaussian Inverse
        for i in range(n):
            # Partial pivoting
            max_row = i
            for j in range(i + 1, n):
                if abs(matrix[j][i]) > abs(matrix[max_row][i]):
                    max_row = j
            matrix[i], matrix[max_row] = matrix[max_row], matrix[i]
            identity[i], identity[max_row] = identity[max_row], identity[i]

            scalar = 1.0 / matrix[i][i]
            for j in range(n):
                matrix[i][j] *= scalar
                identity[i][j] *= scalar

            for j in range(n):
                if i != j:
                    scalar = matrix[j][i]
                    for k in range(n):
                        matrix[j][k] -= scalar * matrix[i][k]
                        identity[j][k] -= scalar * identity[i][k]

        self.__data = [self.__data[0]]
        self.__data[0][2] = 0
        for i in range(self.__data[0][0]):
            for j in range(self.__data[0][1]):
                if identity[i][j] != 0:
                    self[(i, j)] = identity[i][j]

    def determinant(self):
        if self.__data[0][0] != self.__data[0][1]:
            print("Determinant Error: The matrix is not square")
            return None
```

```
          matrix = [[0 for _ in range(self.__data[0][1])] for _ in
range(self.__data[0][0])]
          for i in self.__data[1:]:
               matrix[i[0]][i[1]] = i[2]

          det = 1
          for i in range(self.__data[0][0]):
               maxElem = abs(matrix[i][i])
               maxRow = i
               for k in range(i + 1, self.__data[0][0]):
                    if abs(matrix[k][i]) > maxElem:
                         maxElem = abs(matrix[k][i])
                         maxRow = k
               if i != maxRow:
                    det *= -1
                    matrix[i], matrix[maxRow] = matrix[maxRow], matrix[i]
               det *= matrix[i][i]

               if matrix[i][i] == 0:
                    return 0  # Determinant is 0 if diagonal element becomes 0

               for k in range(i + 1, self.__data[0][0]):
                    c = 0
                    if matrix[k][i] != 0:
                         c = -matrix[k][i] / matrix[i][i]
                         for j in range(i, self.__data[0][0]):
                              matrix[k][j] += c * matrix[i][j]

          return det
```

```
>>> s = """
sp = SparseMatrix(5, 5)
sp[(1, 2)] = 3
sp[(4, 1)] = 1
sp[(2, 0)] = 4
sp[(3, 3)] = 12
"""
>>> s2 = """
sp2 = SparseMatrix(5, 5)
sp2[(1, 3)] = 5
sp2[(4, 1)] = 12
sp2[(2, 0)] = 3
sp2[(3, 4)] = 8
sp2[(3, 1)] = 46
sp2[(2, 2)] = 9
"""
>>> exec(s)
>>> exec(s2)
>>> print(sp)
    0    0    0    0    0
    0    0    3    0    0
    4    0    0    0    0
    0    0    0   12    0
    0    1    0    0    0
>>> print(sp2)
    0    0    0    0    0
    0    0    0    5    0
    3    0    9    0    0
    0   46    0    0    8
    0   12    0    0    0
```

```
>>> sp.display()
+-----+-----+-----+-----+
| Ind | Row | Col | Val |
+-----+-----+-----+-----+
| 1   | 5   | 5   | 4   |
+-----+-----+-----+-----+
| 2   | 2   | 3   | 3   |
| 3   | 3   | 1   | 4   |
| 4   | 4   | 4   | 12  |
| 5   | 5   | 2   | 1   |
+-----+-----+-----+-----+
>>> sp2.display()
+-----+-----+-----+-----+
| Ind | Row | Col | Val |
+-----+-----+-----+-----+
| 1   | 5   | 5   | 6   |
+-----+-----+-----+-----+
| 2   | 2   | 4   | 5   |
| 3   | 3   | 1   | 3   |
| 4   | 3   | 3   | 9   |
| 5   | 4   | 2   | 46  |
| 6   | 4   | 5   | 8   |
| 7   | 5   | 2   | 12  |
+-----+-----+-----+-----+
>>>
```

```
>>> sp3 = sp + sp2
>>> print(sp3)
    0     0     0     0     0
    0     0     3     5     0
    7     0     9     0     0
    0    46     0    12     8
    0    13     0     0     0
>>> sp3.display()
+-----+-----+-----+-----+
| Ind | Row | Col | Val |
+-----+-----+-----+-----+
| 1   | 5   | 5   | 8   |
+-----+-----+-----+-----+
| 2   | 2   | 3   | 3   |
| 3   | 2   | 4   | 5   |
| 4   | 3   | 1   | 7   |
| 5   | 3   | 3   | 9   |
| 6   | 4   | 2   | 46  |
| 7   | 4   | 4   | 12  |
| 8   | 4   | 5   | 8   |
| 9   | 5   | 2   | 13  |
+-----+-----+-----+-----+
>>>
```

Block Matrix

```
class BlockMatrix:
    def __init__(self, size, block_size):
        self.size = size
        self.block_size = block_size
        self.num_blocks = size // block_size
        self.blocks = [[[0] * block_size for _ in range(block_size)] for _ in
range(self.num_blocks ** 2)]

    def insert(self, row, col, value):
        block_row = row // self.block_size
        block_col = col // self.block_size
        inner_row = row % self.block_size
        inner_col = col % self.block_size
        self.blocks[block_row * self.num_blocks +
block_col][inner_row][inner_col] = value

    def display(self):
        for i in range(self.num_blocks):
            for j in range(self.num_blocks):
```

```
                print("Block ({}, {}):".format(i, j))
                for row in range(self.block_size):
                    for col in range(self.block_size):
                        global_row = i * self.block_size + row
                        global_col = j * self.block_size + col
                        if global_row < self.size and global_col < self.size:
                            print(self.blocks[i * self.num_blocks +
j][row][col], end=" ")
                        else:
                            print("0", end=" ")
                    print()
                print()

def conformal_decomposition(matrix):
     length = len(matrix)
     if length != len(matrix[0]):
          print(f"Invalid Dimension Error: Expected a square matrix but a
Matrix({length}x{len(matrix[0])}) was given.")
          return None, None
     diamat = [[0]*length for i in range(length)]
     offdiamat = [[0]*length for i in range(length)]

     for i in range(length):
          for j in range(length):
               if i == j:
                    diamat[i][j] = matrix[i][j]
               else:
                    offdiamat[i][j] = matrix[i][j]
     return diamat, offdiamat


matrix = BlockMatrix(4, 2)
value = 1
for i in range(4):
    for j in range(4):
        matrix.insert(i, j, value)
        value += 1
print("Block Matrix:")
matrix.display()
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print("Conformal Decomposition matrix:")
for i in mat:
     for j in i:
          print("{:>5}".format(j), end="")
     print()
diamat, offdiamat = conformal_decomposition(mat)
print("Diagonal Matrix: ")
for i in diamat:
     for j in i:
          print("{:>5}".format(j), end="")
     print()
```

```
print("Off-Diagonal Matrix: ")
for i in offdiamat:
    for j in i:
        print("{:>5}".format(j), end="")
    print()
```

```
Block Matrix:
Block (0, 0):
1 2
5 6

Block (0, 1):
3 4
7 8

Block (1, 0):
9 10
13 14

Block (1, 1):
11 12
15 16

Conformal Decomposition matrix:
    1    2    3
    4    5    6
    7    8    9
Diagonal Matrix:
    1    0    0
    0    5    0
    0    0    9
Off-Diagonal Matrix:
    0    2    3
    4    0    6
    7    8    0
```