SSCBS

Practical File

DSC 05: Discrete Mathematical Structures

Prepared by :

Avinash Shrivastava

(22512)

Practicals\Q1\Q1.py

```python
# Create a class SET. Create member functions to perform the following SET operations:
# 1) ismember: check whether an element belongs to the set or not and return value as
# true/false.
# 2) powerset: list all the elements of the power set of a set .
# 3) subset: Check whether one set is a subset of the other or not.
# 4) union and Intersection of two Sets.
# 5) complement: Assume Universal Set as per the input elements from the user.
# 6) set Difference and Symmetric Difference between two sets.
# 7) cartesian Product of Sets.
# Write a menu driven program to perform the above functions on an instance of the SET
# class.

class SET():
    def __init__(self, lst) :
        self.set = set(lst)

    def isMember(self,element):
        if element in self.set:
            return True
        return False

    def powerSet(slef):
        pass

    def isSubsetOf(self, otherset):
        for ele in self.set :
            if ele not in otherset.set:
                return False
        return True

    def setUnion(self, otherset):
        unionSet = set()
        for i in self.set:
            unionSet.add(i)
        for j in otherset.set:
            unionSet.add(j)
        return unionSet

    def setIntersection(self, otherset):
        intersect = set()
        for i in self.set:
            if i in otherset.set:
                intersect.add(i)
        for j in otherset.set:
            if j in self.set:
                intersect.add(j)
        return intersect

    def complement(self):
        universalSet = eval(input("Enter Universal Set : "))
        compl = set()
        for i in universalSet:
            if i not in self.set:
                compl.add(i)
        return compl

```

```python
    def setDifference(self, otherset):
        diff = self.set.copy()
        intersection = self.setIntersection(otherset)
        for ele in intersection:
            diff.discard(ele)
        return diff

    def symmetricDifference(self, otherset):
        union = otherset.setUnion(self)
        intersection = otherset.setIntersection(self)
        for i in intersection:
            union.discard(i)
        return union

    def print(self):
        print(self.set)









setA = SET([1,2,3,4,7,12])
setB = SET([4,3,2,1,7,11])

print("setA = ")
setA.print()
print("setB = ")
setB.print()
print("Checking if 11 is member of setA and setB")
print("setA",setA.isMember(11))
print("setB",setB.isMember(11))

print("Union of setA and setB : ")
print(setA.setUnion(setB))

print("Intersection of setA and setB : ")
print(setA.setIntersection(setB))

print("Cheking if setA is subset of setB : ")
print(setA.isSubsetOf(setB))

print("Complement of setA :")
# print(setA.complement())

print("Set Differnece of setA and setB : ")
print(setA.setDifference(setB))

print("Symmentric Difference of setA and setB : ")
print(setA.symmetricDifference(setB))

```

Practicals\Q2\Q2.py

```python
# Q2
# Create a class RELATION, use Matrix notation to represent a relation. Include member
# functions to check if the relation is Reflexive, Symmetric, Anti-symmetric, Transitive.
# Using these functions check whether the given relation is: Equivalence or Partial Order
# relation or None

a = [1,2,3]
rel = [[1,1],[2,2],[3,3]]


class RELATION():

    def __init__(self,rel):
        self.rel = rel
        self.adjMat = self.makeAdjMat(rel)

    def makeAdjMat(self,rel):
        relation = self.rel
        matrix = []
        for i in range(len(a)):
            row = []
            for j in range(len(a)):
                row.append(0)
            matrix.append(row)
        for i in relation:
            row = a.index(i[0])
            col = a.index(i[1])
            matrix[row][col] = 1
        return matrix



    def isReflexive(self):
        for i in range(len(self.adjMat)):
            for j in range(len(self.adjMat)):
                if i == j and self.adjMat[i][j] != 1:
                    return False
        return True

    def isSymmetric(self):
        for i in range(len(self.adjMat)):
            for j in range(len(self.adjMat)):
                if self.adjMat[i][j] == 1 and self.adjMat[j][i] != 1:
                    return False
        return True

    #This function is just for demo purpose. It has very high time complexity hence it is needed to be changed
    def isTransitive(self):
        for i in range(len(rel.adjMat)):
            for j in range(len(rel.adjMat)):
                for k in range(len(rel.adjMat)):
                    if rel.adjMat[i][j] == 1 and rel.adjMat[j][k] == 1 :
                        if rel.adjMat[i][k] != 1:
                            return False
        return True
```

```
    def isEquivalence(self):
        if self.isReflexive() and self.isSymmetric() and self.isTransitive():
            return True
        return False



rel = RELATION([[1,1],[2,2],[3,3],[2,1],[1,2],[2,3]])
print(rel.isReflexive())
print(rel.isSymmetric())
print(rel.isTransitive())
print(rel.isEquivalence())
```

Practicals\Q3\Q3.py

```python
# Write a Program that generates all the permutations of a given set of digits, with or
# without repetition.

from itertools import permutations, combinations_with_replacement

arr = [1, 2, 3, 4]

# Permutations without repetition
perms_without_repetition = list(permutations(arr))
print("Permutations without repetition:")
for perm in perms_without_repetition:
    print(perm)
print()

# Permutations with repetition
perms_with_repetition = list(permutations(arr, len(arr)))
print("Permutations with repetition:")
for perm in perms_with_repetition:
    print(perm)
print()

```

Practicals\Q4\Q4.py

```python
# . For any number n, write a program to list all the solutions of the equation x1 + x2 + x3 +
# ...+ xn = C, where C is a constant (C<=10) and x1, x2,x3,...,xn are nonnegative integers,
# using brute force strategy.

def solve_equation(n, C):
    solutions = []

    def generate_combinations(current_sum, current_combination):
        if current_sum == C and len(current_combination) == n:
            solutions.append(current_combination)
            return
        elif current_sum > C or len(current_combination) > n:
            return

        for i in range(C + 1):
            generate_combinations(current_sum + i, current_combination + [i])

    generate_combinations(0, [])

    return solutions

# Example usage
n = 3
C = 4

solutions = solve_equation(n, C)
print(f"Solutions for n={n}, C={C}: {solutions}")
```

Practicals\Q5\Q5.py

```
#Q5
# Write a Program to evaluate a polynomial function. (For example store f(x) = 4n2 + 2n +
# 9 in an array and for a given value of n, say n = 5, compute the value of f(n)).

def evaluate_polynomial(coefficients, x):
    result = 0
    power = len(coefficients) - 1
    for coefficient in coefficients:
        result += coefficient * (x ** power)
        power -= 1
    return result

# Example usage
polynomial = [4, 2, 9]
x = int(input("Enter value of n : "))  # value of x
result = evaluate_polynomial(polynomial, x)
print(f"The result of evaluating the polynomial at x = {x} is: {result}")
























# n = int(input("Enter value of n : "))
# def calculatePoly(n):
#     poly_fun = 4*n*n + 2*n +9
#     return poly_fun

# print(calculatePoly(n))

# arr = [4*n*n, 2*n, 9]
# print(sum(arr))

# arr [ 1,3,4,5,6]

```

Practicals\Q6\Q6.py

```python
# Write a Program to check if a given graph is a complete graph. Represent the graph using
# the Adjacency Matrix representation.


def is_complete_graph(adjacency_matrix):
    num_vertices = len(adjacency_matrix)

    # Check if each pair of vertices is connected
    for i in range(num_vertices):
        for j in range(num_vertices):
            if i != j and not adjacency_matrix[i][j]:
                return False

    return True

# Example usage
graph = [
    [0, 1, 1, 1],  # Vertex 1 is connected to vertices 2, 3, and 4
    [1, 0, 1, 1],  # Vertex 2 is connected to vertices 1, 3, and 4
    [1, 1, 0, 1],  # Vertex 3 is connected to vertices 1, 2, and 4
    [1, 1, 1, 0]   # Vertex 4 is connected to vertices 1, 2, and 3
]

result = is_complete_graph(graph)
if result:
    print("The graph is a complete graph.")
else:
    print("The graph is not a complete graph.")
```

Practicals\Q7\Q7.py

```python
# Write a Program to check if a given graph is a complete graph. Represent the graph using
# the Adjacency List representation.

def is_complete_graph(adjacency_list):
    num_vertices = len(adjacency_list)

    # Check if each pair of vertices is connected
    for i in range(1, num_vertices + 1):
        for j in range(1, num_vertices + 1):
            if i != j and j not in adjacency_list[i]:
                return False

    return True

# Example usage
graph = {
    1: [2, 3, 4],  # Vertex 1 is connected to vertices 2, 3, and 4
    2: [1, 3, 4],  # Vertex 2 is connected to vertices 1, 3, and 4
    3: [1, 2, 4],  # Vertex 3 is connected to vertices 1, 2, and 4
    4: [1, 2]   # Vertex 4 is connected to vertices 1, 2, and 3
}

result = is_complete_graph(graph)
if result:
    print("The graph is a complete graph.")
else:
    print("The graph is not a complete graph.")
```

Practicals\Q8\Q8.py

```python
# Write a Program to accept a directed graph G and compute the in-degree and out-degree
# of each vertex.


def compute_degrees(graph):
    degrees = {}

    # Initialize degrees dictionary with all vertices
    for vertex in graph:
        degrees[vertex] = {'in_degree': 0, 'out_degree': 0}

    # Compute in-degree and out-degree for each vertex
    for vertex in graph:
        for adjacent_vertex in graph[vertex]:
            # Increment out-degree of the current vertex
            degrees[vertex]['out_degree'] += 1
            # Increment in-degree of the adjacent vertex
            degrees[adjacent_vertex]['in_degree'] += 1

    return degrees

# Example usage
graph = {
    'A': ['B', 'C', 'D'],  # Vertex A has outgoing edges to B, C, D
    'B': ['C', 'D'],       # Vertex B has outgoing edges to C, D
    'C': ['D'],            # Vertex C has an outgoing edge to D
    'D': []                # Vertex D has no outgoing edges
}

degrees = compute_degrees(graph)
for vertex in degrees:
    in_degree = degrees[vertex]['in_degree']
    out_degree = degrees[vertex]['out_degree']
    print(f"Vertex {vertex}: In-Degree = {in_degree}, Out-Degree = {out_degree}")

```

Practicals\thankyou.txt

```
Thank you for spending time and going through these programs!

You can find all these practicals on my Github profile at this link :
https://github.com/AvinashShrivastav/Discrete-mathematical-structure/tree/main/Practicals
```