Discrete Math Problem Solution

# Problem 1: Relations R1 and R2 on Set A = {1,2,3,4}

def find\_relation\_pairs(set\_A):

# R1: {(a,b) | a divides b}

R1 = [(a, b) for a in set\_A for b in set\_A if b % a == 0]

# R2: {(a,b) | a ≤ b}

R2 = [(a, b) for a in set\_A for b in set\_A if a <= b]

print("R1 (a divides b):", R1)

print("R2 (a ≤ b):", R2)

# Test the function

A = [1, 2, 3, 4]

find\_relation\_pairs(A)

# Problem 2: Relation from Set A to Set B with specific conditions

def find\_relation\_matrix(A, B, a1, a2, a3, b1, b2):

# Create the relation R

R = []

for a in A:

for b in B:

if a > b:

R.append((a, b))

# Create the relation matrix

relation\_matrix = [[0 for \_ in range(len(B))] for \_ in range(len(A))]

for a, b in R:

row\_index = A.index(a)

col\_index = B.index(b)

relation\_matrix[row\_index][col\_index] = 1

print("Relation R:", R)

print("Relation Matrix:")

for row in relation\_matrix:

print(row)

# Test the function with given conditions

A = [1, 2, 3]

B = [1, 2]

a1, a2, a3 = 1, 2, 3

b1, b2 = 1, 2

find\_relation\_matrix(A, B, a1, a2, a3, b1, b2)

# Problem 3: Graph Coloring by Welch-Powell's Algorithm

def welch\_powell\_coloring(graph):

# Sort vertices by degree in descending order

vertices = sorted(graph.keys(), key=lambda x: len(graph[x]), reverse=True)

# Initialize colors

colors = {}

color\_count = 0

while vertices:

# Assign a new color

color\_count += 1

colored\_vertices = []

# Color the first uncolored vertex

first\_vertex = vertices[0]

colors[first\_vertex] = color\_count

colored\_vertices.append(first\_vertex)

# Try to color other uncolored vertices

for vertex in vertices[1:]:

# Check if this vertex can be colored with the current color

if all(colors.get(adj, 0) != color\_count for adj in graph[vertex]):

colors[vertex] = color\_count

colored\_vertices.append(vertex)

# Remove colored vertices

for v in colored\_vertices:

vertices.remove(v)

return colors

# Example graph representation (adjacency list)

graph = {

'A': ['B', 'C'],

'B': ['A', 'C', 'D'],

'C': ['A', 'B', 'D', 'E'],

'D': ['B', 'C', 'E', 'F'],

'E': ['C', 'D'],

'F': ['D']

}

# Test the algorithm

color\_assignment = welch\_powell\_coloring(graph)

print("Graph Coloring:")

for vertex, color in color\_assignment.items():

print(f"Vertex {vertex}: Color {color}")

# Problem 4: Shortest Path by Warshall's Algorithm

def warshalls\_algorithm(adjacency\_matrix):

# Get the number of vertices

n = len(adjacency\_matrix)

# Create a copy of the adjacency matrix

dist = [row[:] for row in adjacency\_matrix]

# Warshall's algorithm

for k in range(n):

for i in range(n):

for j in range(n):

# If k is an intermediate vertex on the shortest path from i to j

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

# Example adjacency matrix (use a large value for no direct connection)

INF = float('inf')

graph = [

[0, 5, INF, 10],

[INF, 0, 3, INF],

[INF, INF, 0, 1],

[INF, INF, INF, 0]

]

# Test the algorithm

shortest\_paths = warshalls\_algorithm(graph)

print("Shortest Path Matrix:")

for row in shortest\_paths:

print(row)

# Problem 5: Matrix Operations for Relations M(R1∪R2) and M(R1∩R2)

import numpy as np

def matrix\_union\_intersection(MR1, MR2):

# Convert to numpy arrays for easier matrix operations

MR1 = np.array(MR1)

MR2 = np.array(MR2)

# Matrix Union

M\_union = np.logical\_or(MR1, MR2).astype(int)

# Matrix Intersection

M\_intersection = np.logical\_and(MR1, MR2).astype(int)

print("M(R1 ∪ R2):")

print(M\_union)

print("\nM(R1 ∩ R2):")

print(M\_intersection)

# Given matrices

MR1 = [

[1, 0, 1],

[0, 1, 0],

[0, 0, 1]

]

MR2 = [

[1, 0, 1],

[0, 1, 0],

[1, 0, 1]

]

# Test the function

matrix\_union\_intersection(MR1, MR2)

# Problem 6: Newton-Gregory Forward Interpolation for Population

def newton\_gregory\_forward(x, years, populations, target\_year):

# Number of data points

n = len(years)

# Calculate forward difference table

diff\_table = [[0 for \_ in range(n)] for \_ in range(n)]

# First column is the original populations

for i in range(n):

diff\_table[i][0] = populations[i]

# Calculate forward differences

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i+1][j-1] - diff\_table[i][j-1]

# Calculate u and interpolated value

h = years[1] - years[0] # uniform interval

u = (target\_year - years[0]) / h

# Calculate interpolated population

population = diff\_table[0][0]

u\_term = 1

factorial = 1

for j in range(1, n):

u\_term \*= (u - j + 1)

factorial \*= j

population += (u\_term / factorial) \* diff\_table[0][j]

return population

# Given data

years = [1911, 1921, 1931, 1941, 1951, 1961]

populations = [12, 15, 20, 27, 39, 52]

# Interpolate for 1946

target\_year = 1946

interpolated\_population = newton\_gregory\_forward(len(years), years, populations, target\_year)

print(f"Interpolated population in {target\_year}: {interpolated\_population:.2f}")

# Problem 7: Newton-Gregory Backward Interpolation

def newton\_gregory\_backward(x, x\_values, fx\_values, target\_x):

# Number of data points

n = len(x\_values)

# Calculate backward difference table

diff\_table = [[0 for \_ in range(n)] for \_ in range(n)]

# First column is the original function values

for i in range(n):

diff\_table[i][0] = fx\_values[i]

# Calculate backward differences

for j in range(1, n):

for i in range(n - j):

diff\_table[i][j] = diff\_table[i+1][j-1] - diff\_table[i][j-1]

# Calculate u and interpolated value

h = x\_values[1] - x\_values[0] # uniform interval

u = (target\_x - x\_values[-1]) / h

# Calculate interpolated value

value = diff\_table[-1][0]

u\_term = 1

factorial = 1

for j in range(1, n):

u\_term \*= (u + j - 1)

factorial \*= j

value += (u\_term / factorial) \* diff\_table[-1][j]

return value

# Given data

x\_values = [1, 2, 3, 4, 5, 6, 7, 8]

fx\_values = [1, 8, 27, 64, 125, 216, 343, 512]

target\_x = 5.5

# Calculate interpolated value

interpolated\_value = newton\_gregory\_backward(len(x\_values), x\_values, fx\_values, target\_x)

print(f"Interpolated value at x = {target\_x}: {interpolated\_value:.2f}")

# Problem 8: Newton's Divided Difference Interpolation Formula

def newton\_divided\_difference(x\_values, fx\_values, target\_x):

# Number of data points

n = len(x\_values)

# Create divided difference table

divided\_diff = [[0 for \_ in range(n)] for \_ in range(n)]

# First column is the original function values

for i in range(n):

divided\_diff[i][0] = fx\_values[i]

# Calculate divided differences

for j in range(1, n):

for i in range(n - j):

divided\_diff[i][j] = (divided\_diff[i+1][j-1] - divided\_diff[i][j-1]) / (x\_values[i+j] - x\_values[i])

# Interpolation calculation

result = divided\_diff[0][0]

product\_term = 1

for j in range(1, n):

product\_term \*= (target\_x - x\_values[j-1])

result += divided\_diff[0][j] \* product\_term

return result

# Given data

x\_values = [4, 5, 7, 10, 11, 13]

fx\_values = [48, 100, 294, 900, 1210, 2028]

target\_x = 15

# Calculate interpolated value

interpolated\_value = newton\_divided\_difference(x\_values, fx\_values, target\_x)

print(f"Interpolated value at x = {target\_x}: {interpolated\_value:.2f}")

# Problem 9: Lagrange's Interpolation Formula for Unequal Intervals

def lagrange\_interpolation(x\_values, y\_values, target\_x):

# Number of data points

n = len(x\_values)

# Initialize interpolated value

interpolated\_y = 0

# Lagrange interpolation formula

for i in range(n):

# Calculate Lagrange basis polynomial

basis\_poly = 1

for j in range(n):

if i != j:

basis\_poly \*= (target\_x - x\_values[j]) / (x\_values[i] - x\_values[j])

# Multiply basis polynomial with corresponding y value

interpolated\_y += y\_values[i] \* basis\_poly

return interpolated\_y

# Given data

x\_values = [5, 6, 9, 11]

y\_values = [12, 13, 14, 16]

target\_x = 10

# Calculate interpolated value

interpolated\_y = lagrange\_interpolation(x\_values, y\_values, target\_x)

print(f"Interpolated y value at x = {target\_x}: {interpolated\_y:.2f}")

# Problem 10: Bisection Method to Find Real Root

def bisection\_method(f, a, b, tolerance=1e-6, max\_iterations=100):

# Check if root is bracketed

if f(a) \* f(b) >= 0:

print("Bisection method fails: No sign change between a and b")

return None

# Iterations

for iteration in range(max\_iterations):

# Calculate midpoint

c = (a + b) / 2

fc = f(c)

# Print iteration details

print(f"Iteration {iteration + 1}: a = {a}, b = {b}, c = {c}, f(c) = {fc}")

# Check if midpoint is the root

if abs(fc) < tolerance:

print(f"Root found: {c}")

return c

# Update interval

if f(a) \* fc < 0:

b = c

else:

a = c

# If max iterations reached

print("Maximum iterations reached")

return (a + b) / 2

# Define the function x^2 - 4x - 10 = 0

def f(x):

return x\*\*2 - 4\*x - 10

# Solve for root between -2 and -1.5

root = bisection\_method(f, -2, -1.5)

if root is not None:

print(f"Approximate root: {root}")

print(f"Function value at root: {f(root)}")

# Problem 11: False Position Method to Find Root

def false\_position\_method(f, a, b, tolerance=1e-6, max\_iterations=100):

# Check if root is bracketed

if f(a) \* f(b) >= 0:

print("False Position method fails: No sign change between a and b")

return None

# Iterations

for iteration in range(max\_iterations):

# Calculate intersection point with x-axis using False Position formula

c = (a \* f(b) - b \* f(a)) / (f(b) - f(a))

fc = f(c)

# Print iteration details

print(f"Iteration {iteration + 1}: a = {a}, b = {b}, c = {c}, f(c) = {fc}")

# Check if root is found

if abs(fc) < tolerance:

print(f"Root found: {c}")

return c

# Update interval

if f(a) \* fc < 0:

b = c

else:

a = c

# If max iterations reached

print("Maximum iterations reached")

return (a + b) / 2

# Define the function x^2 - x - 2 = 0

def f(x):

return x\*\*2 - x - 2

# Solve for root between 1 and 3

root = false\_position\_method(f, 1, 3)

if root is not None:

print(f"Approximate root: {root}")

print(f"Function value at root: {f(root)}")