M.Sc. (Five Year Integrated) in Computer Science (Artificial Intelligence & Data Science)

Fourth Semester

Laboratory Record

21-805-0407: OPTIMIZATION TECHNIQUES LAB

Submitted in partial fulfillment
of the requirements for the award of degree in
Master of Science (Five Year Integrated)
in Computer Science (Artificial Intelligence & Data Science) of
Cochin University of Science and Technology (CUSAT)
Kochi

Submitted by

K A GANESH KUMAR
(80522011)

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI-682022

MAY 2024

DEPARTMENT OF COMPUTER SCIENCE

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)

KOCHI, KERALA-682022

*This is to certify that the software laboratory record for **21-805-0407: Optimization Techniques Lab** is a record of work carried out by **K A GANESH KUMAR (80522011)**in partial fulfillment of the requirements for the award of degree in **Master of Science (Five Year Integrated)** in **Computer Science (Artificial Intelligence & Data Science)** of Cochin University of Science and Technology (CUSAT), Kochi. The lab record has been approved as it satisfies the academic requirements in respect of the third semester laboratory prescribed for the Master of Science (Five Year Integrated) in Computer Science degree.*

Faculty Member in-charge

Dr. Remya K Sasi	Dr. Madhu S Nair
Assistant Professsor	Professor and Head
Department of Computer Science	Department of Computer Science
CUSAT	CUSAT

Table of Contents

SIMPLEX METHOD

AIM

To implement and understand the simplex method for solving linear programming problems and finding the optimal solution by iteratively moving towards the optimal vertex of the feasible region.

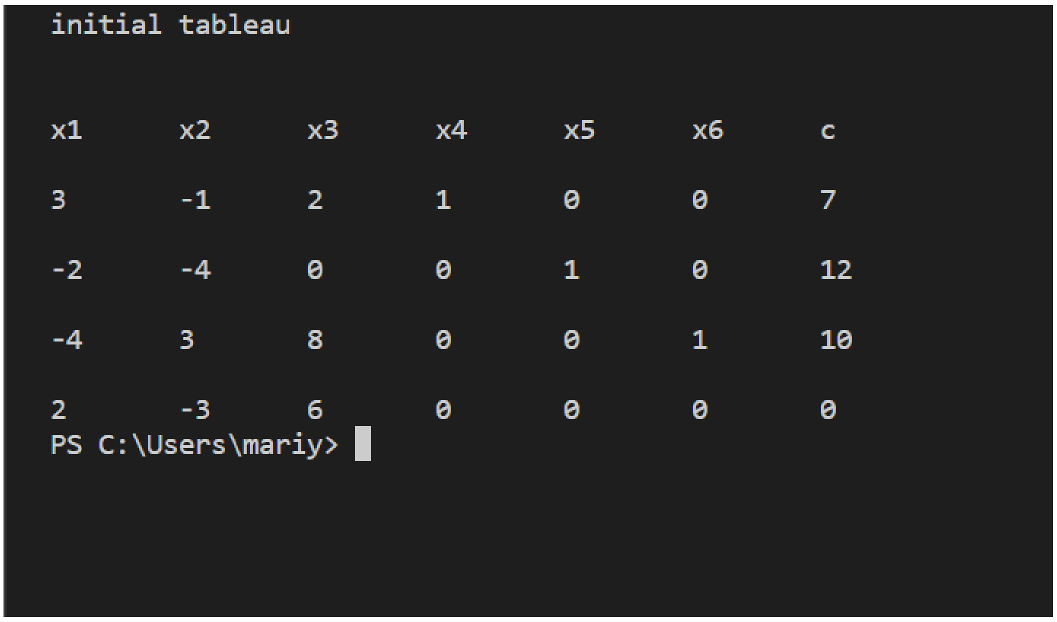
PROGRAM

```
#Simplex method

import math
import numpy as np
from fractions import Fraction




# Convertion into tableau

def to_tableau(z,A,b):
    # Augmenting A and b
    tableau = [eq + [x] for eq, x in zip(A, b)]
    # Adding 0 to make z size compatible to A|b
    z = z + [0]
    #Adding z as the final row
    tableau += [z]
    return tableau

# Function to print tableau
def print_table(tableau):
    n_var = len(tableau[0])
    var = ["x{0}".format(i+1) for i in range(n_var-1)]
    for j in var:
        print(j,end = '\t')
    print("c ",end = '\n\n')

    for k in tableau:
        for eq in k:
            print(Fraction(str(eq)).limit_denominator(100), end ='\t')
        print('\n')
```

```
# Function to check for optimality
def is_optimal(tableau):
    # Optimality is reached only when z < 0
    z = tableau[-1]
    # z[-1] always equals 0
    # any() would return a boolean value
    return not any(i > 0 for i in z[:-1])

# Function to find pivot element
def find_pivot_element(tableau):
    '''
    pivot column (pc) is the maximum value of z
    pivot row (pr) is the min of ratios of b values with pivot column values
    pr = index(min( [ b[i] / pc[i] ]))
    '''
    z = tableau[-1]
    ipc = next(i for i, x in enumerate(z[:-1]) if x > 0)
    ratio = []
    for eq in tableau[:-1]:
        el = eq[ipc]
        ratio.append(math.inf if el < 0 else eq[-1] / el)
    ipr = ratio.index(min(ratio))
    return ipr,ipc



def pivot_step(tableau,pivot_position):
    # finding zj
    new_tableau = [[] for i in tableau]
    ipr,ipc = pivot_position
    pivot_element = tableau[ipr][ipc]
    # Dividing the pivot row by pivot element
    new_tableau[ipr] = [el/pivot_element for el in tableau[ipr]]
    # Changing the other rows
    for eq_i,eq in enumerate(tableau):
        if eq_i != ipr:
            new_tableau[eq_i] = [
                tableau[eq_i][i] - new_tableau[ipr][i] * tableau[eq_i][ipc]
                for i in range(len(new_tableau[ipr]))
                ]
    return new_tableau
```

```
def simplex(z,A,b):
    # Converting the equations to tableau
    tableau = to_tableau(z,A,b)
    print("initial tableau\n\n")
    print_table(tableau)
    find_pivot_element(tableau)
    it = 1
    while not is_optimal(tableau):
        pivot_position = find_pivot_element(tableau)
        tableau = pivot_step(tableau,pivot_position)
        print('\n\niteration: {0}\n'.format(it))
        print_table(tableau)
        it+=1
    return tableau



def is_basic(column):
    return sum(column) == 1 and len([c for c in column if c == 0]) == len(column) - 1

def get_solution(tableau):
    columns = np.array(tableau).T
    solutions = []
    for column in columns[:-1]:
        solution = 0
        if is_basic(column):
            one_index = column.tolist().index(1)
            solution = columns[-1][one_index]
        solutions.append(solution)
    return solutions

def show_solutions(tableau):
    solutions = get_solution(tableau)
    var = ["x{0}".format(i+1) for i in range(len(tableau)-1)]
    for s,v in zip(solutions,var):
        print("{0} : {1}".format(v,Fraction(str(s)).limit_denominator(100)),end = '\n')
    print("z : {0}".format(Fraction(str(tableau[-1][-1])).limit_denominator(100)))

def main():

    '''
    Consider the LPP

    maximize z = 12x1 + 16x2
    ST:
    1) 10x1 + 20x2 + x3 = 120
    2)  8x1 +  8x2 + x4 = 80
    3)  x1,x2,x3,x4 > 0
    '''
    z = [2,-3,6,0,0,0]
    A = [
            [ 3,-1, 2, 1, 0, 0],
            [-2,-4, 0, 0, 1, 0],
            [-4, 3, 8, 0, 0, 1]
        ]
    b = [ 7,12,10]
    show_solutions(simplex(z,A,b))

if __name__ == "__main__":
    main()
```

SAMPLE INPUT-OUTPUT



```
initial tableau


x1      x2      x3      x4      x5      x6      c

3       -1      2       1       0       0       7

-2      -4      0       0       1       0       12

-4      3       8       0       0       1       10

2       -3      6       0       0       0       0
PS C:\Users\mariy>
```

DUAL SIMPLEX METHOD

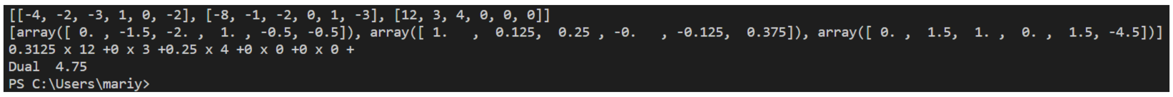
AIM

To implement and explore the dual simplex method, an extension of the simplex method, for solving linear programming problems with degenerate or infeasible solutions by iteratively improving the dual feasibility.

PROGRAM

```
import numpy as np

c = [12, 3, 4, 0, 0]
A = [
    [-4, -2, -3,  1,  0],
    [-8, -1, -2,  0,  1]
]
b = [-2, -3]



def pivot_step(tableau, pivot_position):
    new_tableau = [[] for eq in tableau]

    i, j = pivot_position
    pivot_value = tableau[i][j]
    new_tableau[i] = np.array(tableau[i]) / pivot_value

    for eq_i, eq in enumerate(tableau):
        if eq_i != i:
            multiplier = np.array(new_tableau[i]) * tableau[eq_i][j]
            new_tableau[eq_i] = np.array(tableau[eq_i]) - multiplier

    return new_tableau

# Converting to tableau
def to_tableau(c, A, b):
    xb = [eq + [x] for eq, x in zip(A, b)]
    z = c + [0]
    return xb + [z]


def is_basic(column):
    return sum(column) == 1 and len([c for c in column if c == 0]) == len(column) - 1

def get_solution(tableau):
    columns = np.array(tableau).T
    solutions = []
    for column in columns[:-1]:
        solution = 0
        if is_basic(column):
            one_index = column.tolist().index(1)
            solution = columns[-1][one_index]
        solutions.append(solution)
    return solutions

# Find the rows with
def can_be_improved_for_dual(tableau):
    rhs_entries = [row[-1] for row in tableau[:-1]]
    return any([entry < 0 for entry in rhs_entries])

#Finding the pivot position
def get_pivot_position_for_dual(tableau):
    rhs_entries = [row[-1] for row in tableau[:-1]]
    min_rhs_value = min(rhs_entries)
    row = rhs_entries.index(min_rhs_value)

    columns = []
    for index, element in enumerate(tableau[row][:-1]):
        if element < 0:
            columns.append(index)
    columns_values = [tableau[row][c] / tableau[-1][c] for c in columns]
    column_min_index = columns_values.index(min(columns_values))
    column = columns[column_min_index]

    return row, column

def dual_simplex(c, A, b):
    # Conversion to simplex table
    tableau = to_tableau(c, A, b)

    # While
    while can_be_improved_for_dual(tableau):
        print(tableau)
        pivot_position = get_pivot_position_for_dual(tableau)
        tableau = pivot_step(tableau, pivot_position)

    return get_solution(tableau)

soln  = dual_simplex(c, A, b)
opt = 0
for i,x in zip(soln,c):
    print('{} x {} '.format(i,x), end = '+')
    opt+=i*x
print('\nDual ',opt)
```

SAMPLE INPUT-OUTPUT



```
[[-4, -2, -3, 1, 0, -2], [-8, -1, -2, 0, 1, -3], [12, 3, 4, 0, 0, 0]]
[array([ 0. , -1.5, -2. ,  1. , -0.5, -0.5]), array([ 1.    ,  0.125,  0.25 , -0.    , -0.125,  0.375]), array([ 0. ,  1.5,  1. ,  0. ,  1.5, -4.5])]
0.3125 x 12 +0 x 3 +0.25 x 4 +0 x 0 +0 x 0 +
Dual  4.75
PS C:\Users\mariy>
```

TRANSPORTATION PROBLEMS

AIM

To solve the transportation problem using optimization techniques and algorithms, and to determine the optimal way to transport goods from multiple suppliers to multiple consumers while minimizing transportation costs.

PROGRAM

```
#Transportation problems
import numpy as np



'''
Function to find the northwest corner

input: temp_source temp_dest arrays

returns: row and column value of NW corner
'''
def find_north_west_corner(temp_source,temp_dest):
    for i,s in enumerate(temp_source):
        if ( s != 0 ):
            for j,d in enumerate(temp_dest):
                if ( d != 0 ):
                    return (i,j)

    return False

'''
Function to apply northwest corner method

returns: Matrix of the costs for each cell
'''


def north_west_corner(sources,dest,costs):
    # temp_sources and dests are used to dectement costs
    temp_source = sources.copy()
    temp_dest = dest.copy()
    n_source,n_dests = costs.shape
    # To store assigned costs
    assigned = np.zeros(costs.shape)

    while find_north_west_corner(temp_source,temp_dest):
        x,y = find_north_west_corner(temp_source,temp_dest)
        min_cost = min(temp_source[x],temp_dest[y])
        temp_source[x] -= min_cost
        temp_dest[y] -= min_cost
        assigned[x][y] = min_cost
    return assigned

'''
Function to find cost given the assignments
'''
def get_cost(cost,assigned):
    total_cost = 0
    for c_r,a_r in zip(cost,assigned):
        for c,a in zip(c_r,a_r):
            total_cost+=c*a
    return total_cost



'''
Function to find the u and v values

input: sources,dest,costs,assigned
output: u,v values

This function is used to find the u and v values to find the penalty

'''
def find_uv_values(sources,dest,costs,assigned):
    # Finding u,v values to find the penalty
    U = [None for i in range(len(sources))]
    V = [None for i in range(len(dest))]
    U[0] = 0
    # Variable to check if the V and V columns are filled
    filled = False
    while not filled:
        # Iterating through columns to find Vi's
        for i,row in enumerate(assigned):
            for j,row_el in enumerate(row):
                # Only enter if U_i is assigned
                    if U[i] != None:
                    # If an assigned row
                        if row_el > 0 :
                            # If V is not filled
                            if V[j] == None:
                                # Assign new value for V_i
                                V[j] = costs[i][j] - U[i]
        # Iterating the rows to find U_i’s
        assigned_T = np.transpose(np.array(assigned))
        for i,col in enumerate(assigned_T):
            for j,col_el in enumerate(col):
                # Only enter if V_i is assigned
                    if V[i] != None:
                    # If an assigned row
                        if col_el > 0 :
                            # If V is not filled
                            if U[j] == None:
                                # Assign new value for V_i
                                U[j] = costs[i][j] - V[i]

        # Checking if u and v are filled
        filled_u = True
        filled_v = True
        for u in U:
            if u == None:
                filled_u == False
        for v in V:
            if v == None:
                filled_v = False
        if filled_u and filled_v:
            filled = True
        else:
            filled = False
    return U,V

’’’
Function to find the penalty values
P = u_i + v_j - c_ij
```

```
input: U,V,costs
output: penalty value matrix
'''
def get_penalties(U,V,cost,assigned):
    penalty = np.zeros(cost.shape)
    for i,u in enumerate(U):
        for j,v in enumerate(V):
            if assigned[i][j] == 0:
                penalty[i][j] = u+v-cost[i][j]
    return penalty

'''
Function to check if the given penalty matrix is optimal

input: penalty matrix
output: True if optimal else False
'''
def check_optimality(penalty):
    for row in penalty:
        for el in row:
            if el > 0:
                return False
    return True

'''
Finding the entering variable which has the maximum penalty
input: penalty matrix
'''
def get_entering_variable_position(penalty):
    penalty_copy = np.array(penalty)
    penalty_copy = penalty_copy.flatten()
    penalty_copy = np.sort(penalty_copy)
    position = np.where(penalty == penalty_copy[-1])
    return int(position[0]),int(position[1])

def get_next_cells(matrix, position):
    x,y = position
    next_cells = [(x-1,y),(x+1,y),(x,y-1),(x,y+1)]
    next_cells = [
                    cell for cell in next_cells if
                    cell[0] >= 0
                    and cell[1] >= 0
                    and cell[0] < matrix.shape[0]
                    and cell[1] < matrix.shape[1]
                    and matrix[cell[0], cell[1]] != 0
                ]
    return next_cells



'''
Function to find the closed loop to be pivoted

input: cost,assigned,entering_variable

'''

def find_closed_loop(assigned, start, visited, path, check = 0):
    i, j = start
    visited[i, j] = 1

    next_cells = get_next_cells(assigned, start)
    for cell in next_cells:
        if visited[cell[0], cell[1]] == 0:
            if cell in path:
                return path
            else:
                visited[cell[0], cell[1]] = 1
                result = find_closed_loop(assigned, cell, visited, path + [cell],check+1)
                if result is not None:
                    return result
        elif check >= 2:
            return path



def get_closed_loop(assigned, start):
    visited = np.zeros(assigned.shape)
    path = [start]
    final_path = find_closed_loop(assigned, start, visited, path)
    return final_path

'''
Function to find the pivot cell
```

```
'''
def pivoting(cost,assigned,closed_loop):
    # Finding the pivot cell
    pivot_cell = closed_loop[1]
    assigned_copy = np.array(assigned)
    pivot_cost = assigned[pivot_cell[0],pivot_cell[1]]
    for i in range(len(closed_loop)):
        cell = closed_loop[i]
        if i%2 == 0:
            assigned_copy[cell[0],cell[1]] += pivot_cost
        else:
            assigned_copy[cell[0],cell[1]] -= pivot_cost
    return assigned_copy

def display_results(cost_matrix, allocations, u_values, v_values):
    num_sources, num_destinations = cost_matrix.shape
    # Display the cost matrix
    print("Cost Matrix:")
    for i in range(num_sources):
        for j in range(num_destinations):
            print("{:<5}".format(cost_matrix[i, j]), end=" ")
        print()

    print()

    # Display the assigned values
    print("Assigned Values:")
    for i in range(num_sources):
        for j in range(num_destinations):
            print("{:<5}".format(allocations[i, j]), end=" ")
        print()

    print()

    # Display the u values
    print("U Values:")
    for i in range(num_sources):
        print("U[{}]: {}".format(i, u_values[i]))

    print()
    # Display the v values
    print("V Values:")
    for j in range(num_destinations):
        print("V[{}]: {}".format(j, v_values[j]))

    print()
'''
Program to apply Modi method to find optimal solution
1. Find u,v values
2.
'''
def Modi_method(sources,dest,costs):
    # Getting the intial feasibl solution
    assigned  = north_west_corner(sources,dest,costs)
    print("initial assigned values : \n",assigned)
    print("initial cost: ",get_cost(costs,assigned))
    # Defining penalty P
    P = lambda u,v,c : u+v-c
    # Finding u and v values
    U,V = find_uv_values(sources,dest,costs,assigned)
    print("Initial U values: ",U)
    print("Initial V values: ",V)
    # Finding the penalty matrix
    penalty = get_penalties(U,V,costs,assigned)
    print("Initial Penalty matrix: \n",penalty)
    # Checking if the penalty matrix is optimal
    while not check_optimality(penalty):
        start = get_entering_variable_position(penalty)
        loop = get_closed_loop(assigned,start)
        assigned = pivoting(costs,assigned,loop)
        U,V = find_uv_values(sources,dest,costs,assigned)
        penalty = get_penalties(U,V,costs,assigned)
        display_results(costs,assigned,U,V)

    print("Final assigned values : \n",assigned)




if __name__ == "__main__":
    # input datas
```

```
sources = np.array([250,350,400])
dest = np.array([200,300,350,150])

costs = np.array([
        [3,1,7,4],
        [2,6,5,9],
        [8,3,3,2]
    ])

Modi_method(sources,dest,costs)
```

SAMPLE INPUT-OUTPUT

ASSIGNMENT PROBLEM

AIM

To apply optimization methods to solve the assignment problem, which involves finding the optimal assignment of a set of tasks to a set of workers with minimum total cost or maximum total benefit.

PROGRAM

```
#Assignment Problem

import numpy as np

'''
To reduce smallest element from each row and column
'''
def reduction(cost):

    num_rows, num_columns = cost.shape
    # Reducing from rows
    for i,row in enumerate(cost):
        min_el = np.min(row)
        for j,el in enumerate(row):
            cost[i][j] -= min_el
    # Reducing from columns
    for k in range(num_columns):
        # getting kth column
        column = cost[:,k]
        min_el = np.min(column)
        for l in range(num_rows):
            cost[l][k] -= min_el
    return cost

'''
function to perform assignment given reduced cost matrix
'''
def assignment(cost):
    # assignment Matrix with cross and circles
    # CROSS = -1 denotes unassigned 0
    # CIRCLE = 1 denotes an assigned 0
    assigned = np.zeros(cost.shape)
    # Traverse via rows
    for i,row in enumerate(cost):
        zeros = np.where(row == 0)
        assigned_row = assigned[i,:]
        unmarked = False
        # Check for single unmarked zero
        for z in zeros[0]:
            if assigned_row[z] == 0:
                unmarked = True
        if unmarked or len(zeros[0]) == 1:
            # assign the 0
            assigned[i,zeros[0][0]] = 1
            # Selecting column with the 0
            column = cost[:,zeros[0][0]]
            # Select zeros in the column
            col_zeros = np.where(column == 0)
            # Cross the remaining zeros in the column
            for index in col_zeros[0]:
                if index != i:
                    assigned[index,zeros[0][0]] = -1
    # Traverse via columns
    num_rows, num_columns = cost.shape
    for j in range(num_columns):
        column = cost[:,j]
        zeros = np.where(column == 0)
        assigned_column = assigned[:,j]
        unmarked = False
        # Check for single unmarked zero
        for z in zeros[0]:
            if assigned_column[z] == 0:
                unmarked = True
        if unmarked or len(zeros[0]) == 1:
            # assign the 0
            assigned[zeros[0][0],j] = 1
            # Selecting row with the 0
            row = cost[zeros[0][0],:]
            # Select zeros in the row
            row_zeros = np.where(row == 0)
            # Cross the remaining zeros in the row
            for index in row_zeros[0]:
                if index != j:
                    assigned[zeros[0][0],index] = -1
    return assigned
'''
Function to check if the assignment is optimal

It checks if all the rows and columns have been assigned
'''


def is_optimal(assigned):
    num_assigned = np.where(assigned == 1)
    if len(num_assigned[0]) == assigned.shape[0]:
        return True
    else:
        return False

'''
Function to mark matrix to get minimal lines via all zeros

marking conditions:
1. If a row has no assigned zeros, mark the row
2. columns with 0 in the marked rows, mark the column
3. rows with assigned 0 in the marked columns, mark the row
'''
def mark_assigned_martrix(assigned):
    mark_row = np.zeros(assigned.shape[0])
    mark_column = np.zeros(assigned.shape[1])
    # Marking rows without assigned zeros
    for i,row in enumerate(assigned):
        zeros = np.where(row == 1)
        if len(zeros[0]) == 0:
            mark_row[i] = 1
    # Marking columns with zeros in marked rows
    for j,mark in enumerate(mark_row):
        # in case of a marked row
        if mark == 1:
            row = assigned[j]
            for r,el in enumerate(row):
                # presence of 0
                if el == 1 or el == -1:
                    mark_column[r] = 1
    # Marking rows with assigned 0 in the marked columns
    for k,mark in enumerate(mark_column):
        # if a marked column
        if mark == 1 :
            column = assigned[:,k]
            for c,mark in enumerate(column):
                # in the case of an assigned 0
                if mark == 1:
                    mark_row[c] = 1
    return mark_row,mark_column

'''
Function to find a new cost matrix given the marked rows and columns

Lines are drawn via unmarked rows and marked columns
1. Find smallest element in the non covered elements
2. Subtract the smallest element from all non covered
3. Add it to interesecting lines
'''
def get_new_cost_matrix(cost,assigned,mark_row,mark_column):
    line_column = mark_column.copy()
    line_row = mark_row.copy()
    # unmarked rows
    line_row = 1 - line_row
    # Finding the smallest uncovered element
    uncovered = []
    for i,row in enumerate(assigned):
        for j,col in enumerate(row):
            if line_column[j] != 1 and line_row[i] != 1:
                uncovered.append(cost[i][j])
    min_el = min(uncovered)
    # Subtracting and adding min_el
    for i,row in enumerate(cost):
        for j,col in enumerate(row):
            if line_column[j] != 1 and line_row[i] != 1:
                cost[i][j] -= min_el
            elif line_column[j] == 1 and line_row[i] == 1:
                cost[i][j] += min_el
    return cost
'''
Function to get the final solution given assignment matrix
```

```
'''
def get_solution(assigned):
    solution = []
    for i,row in enumerate(assigned):
        for j,col in enumerate(row):
            if col == 1:
                solution.append((i,j))
    return solution

# To print the matrix
def print_matrix(matrix):
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            print("{:<5}".format(matrix[i, j]), end=" ")
        print()
'''
Hungarian assignment

input: cost 2x2 array
output: assigned elements
'''
def hungarian_assignment(cost):
    print("Initial cost :\n ")
    print_matrix(cost)
    cost = reduction(cost)
    print("\nReduced Cost: \n")
    print_matrix(cost)
    assigned = assignment(cost)
    print("\nIntitial assignement: \n")
    print_matrix(assigned)
    iteration = 0
    # Iterating till optimality is reached
    while not is_optimal(assigned):
        print("\n\niteration {0}".format(iteration))
        print("\nNot optimal")
        # Marking rows and columns
        mark_row, mark_column = mark_assigned_martrix(assigned)
        print("\nMarked rows: ",mark_row)
        print("\nMarked columns: ",mark_column)
        # Getting new cost matrix usign marked rows and columns
        cost = get_new_cost_matrix(cost,assigned,mark_row,mark_column)
        print("\nNew cost : \n")
        print_matrix(cost)
        # Assigning again for the new cost matrix
        assigned = assignment(cost)
        print("\nNew assignment: \n")
        print_matrix(assigned)
        iteration += 1
    # Getting the solution
    print("Optimality reached")
    print("\nFinal assigned matrix: \n")
    print_matrix(assigned)
    solution = get_solution(assigned)
    print("Solution: ",solution)

if __name__ == "__main__":
    cost = np.array([
        [85, 75, 65, 125, 75],
        [90, 78, 66, 132, 78],
        [75, 66, 57, 114, 69],
        [80, 72, 60, 120, 72],
        [76, 64, 56, 112, 68]
    ])
    hungarian_assignment(cost)
    print("Done!")
```

SAMPLE INPUT-OUTPUT

TRAVELLING SALESMAN PROBLEM

AIM

To explore various approaches to solving the traveling salesman problem, a classic optimization problem where the objective is to find the shortest possible route that visits a set of cities and returns to the starting city.

PROGRAM

```
 #Traveling Salesman Problem

import numpy as np

def calculate_cost_matrix(distances):
    # Subtract the distances from the maximum distance
    # to convert the problem into a minimization problem
    max_distance = np.max(distances)
    cost_matrix = max_distance - distances
    return cost_matrix

def solve_tsp(distances,min_element = 0):
    cost_matrix = calculate_cost_matrix(distances)
    n = cost_matrix.shape[0]

    # Step 1: Subtract the minimum value in each row
    min_rows = np.min(cost_matrix, axis=1)
    for i in range(n):
        cost_matrix[i, :] -= min_rows[i]

    # Step 2: Subtract the minimum value in each column
    min_cols = np.min(cost_matrix, axis=0)
    for i in range(n):
        cost_matrix[:, i] -= min_cols[i]

    # Step 3: Cover the zeros with the minimum number of lines
    row_covered = np.zeros(n, dtype=bool)
    col_covered = np.zeros(n, dtype=bool)

    while True:
        zeros = np.where(cost_matrix == 0)
        row_zeros, col_zeros = zeros[0], zeros[1]
        num_zeros = len(row_zeros)

        if num_zeros >= n:
            break

        for i in range(num_zeros):
            row = row_zeros[i]
            col = col_zeros[i]

            if not row_covered[row] and not col_covered[col]:
                row_covered[row] = True
                col_covered[col] = True
                break

        # Step 4: Create additional zeros
        while True:
            cols_covered = np.where(row_covered)[0]
            covered_rows = np.where(col_covered)[0]

            if len(cols_covered) + len(covered_rows) >= n:
                break

            uncovered_rows = np.where(~row_covered)[0]
            uncovered_cols = np.where(~col_covered)[0]
            min_val = np.min(cost_matrix[uncovered_rows, :][:, uncovered_cols])

            for row in uncovered_rows:
                cost_matrix[row, uncovered_cols] -= min_val

            for col in uncovered_cols:
                cost_matrix[uncovered_rows, col] -= min_val

    # Step 5: Find the optimal assignment
    assignment = np.zeros(n, dtype=int)
    rows, cols = np.where(cost_matrix == 0)

    for i in range(len(rows)):
        row, col = rows[i], cols[i]

        if assignment[col] == 0:
            assignment[col] = row
    return assignment

def is_solution_valid(solution):
    visited_nodes = set()
    for node in solution:
        if node in visited_nodes:
            return False
        visited_nodes.add(node)
    return len(visited_nodes) == len(solution)

# Example usage
# distances = np.array([[0, 2, 9, 10],
#                       [1, 0, 6, 4],
#                       [15, 7, 0, 8],
#                       [6, 3, 12, 0]])
distances =np.array( [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]

optimal_order = solve_tsp(distances)
cost = distances[optimal_order, range(distances.shape[1])].sum()
print(optimal_order)
print(cost)
```

TRAVELLING SALESMAN PROBLEM 2

```
from sys import maxsize
from itertools import permutations
V = 4

def tsp(graph, s):
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    min_cost = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        current_cost = 0
        k = s
        for j in i:
            current_cost += graph[k][j]
            k = j
        current_cost += graph[k][s]
    min_cost = min(min_cost, current_cost)
    return min_cost , vertex

graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
distances = [[0, 2, 9, 10],
             [1, 0, 6, 4],
             [15, 7, 0, 8],
             [6, 3, 12, 0]
             ]
s = 0
min_cost,vertex = tsp(distances, s)
print(min_cost)

print("Path: 0",end=" ")
for i in vertex:
    print(i,end=" ")
```

SAMPLE INPUT-OUTPUT

SEQUENCING PROBLEM

AIM

To apply optimization techniques to solve sequencing problems, such as job scheduling, task sequencing, or order processing, with the goal of minimizing completion time, maximizing throughput, or meeting other specified criteria.

PROGRAM

```
#Sequencing Problem

import numpy as np

def sequence_assign(seq_table):
    if len(seq_table) != 2:
        seq_table = convert_to_2_machines(seq_table)
    print("seq_table", seq_table)
    assign = np.zeros(len(seq_table[0]))
    marked = np.zeros([2, len(seq_table[0])])
    i_H = 0
    i_G = -1

    while (np.sum(marked) / 2) != len(seq_table[0]):
        print("marked", marked)
        print("zeros", np.min(seq_table[marked == 0]))
        print(seq_table)

        min_value = np.min(seq_table[marked == 0])  # Corrected line
        min_index = np.where(seq_table == min_value)
        print("min_index", min_index)

        if len(min_index[1]) == 1:
            if min_index[0][0] == 0:
                assign[i_H] = min_index[1][0]
                marked[0, min_index[1][0]] = 1
                marked[1, min_index[1][0]] = 1
                i_H += 1
            elif min_index[0][0] == 1:
                assign[i_G] = min_index[1][0]
                marked[0, min_index[1][0]] = 1
                marked[1, min_index[1][0]] = 1
                i_G -= 1
        else:
            if min_index[0][0] == min_index[0][1]:
                if seq_table[min_index[0][1], min_index[1][1]]
                < seq_table[min_index[0][0], min_index[1][0]]:
                    assign[i_G] = min_index[1][1]
                    marked[0, min_index[1][1]] = 1
                    marked[1, min_index[1][1]] = 1
                    i_G -= 1
                else:
                    assign[i_H] = min_index[1][0]
                    marked[0, min_index[1][0]] = 1
                    marked[1, min_index[1][0]] = 1
                    i_H += 1
            else:
                assign[i_H] = min_index[1][0]
                assign[i_G] = min_index[1][1]
                marked[0, min_index[1][0]] = 1
                marked[1, min_index[1][1]] = 1
                i_H += 1
                i_G -= 1

    return assign, marked

def main():
    table = np.array([[2, 5, 4, 9, 6, 8, 7, 5, 4], [6, 8, 7, 4, 3, 9, 3, 8, 11]])
    assign, marked = sequence_assign(table)
    print("Assignments:", assign)
    print("Marked:", marked)

if __name__ == "__main__":
    main()
```

SAMPLE INPUT-OUTPUT

INTEGER PROGRAMMING PROBLEM

AIM

To study and implement integer programming techniques for solving optimization problems where some or all of the decision variables must take integer values, and to find solutions that satisfy both linear constraints and integrality constraints.

PROGRAM

```
#Simplex method

import math
import numpy as np
from fractions import Fraction




# Convertion into tableau

def to_tableau(z,A,b):
    # Augmenting A and b
    tableau = [eq + [x] for eq, x in zip(A, b)]
    # Adding 0 to make z size compatible to A|b
    z = z + [0]
    #Adding z as the final row
    tableau += [z]
    return tableau

# Function to print tableau
def print_table(tableau):
    n_var = len(tableau[0])
    var = ["x{0}".format(i+1) for i in range(n_var-1)]
    for j in var:
        print(j,end = '\t')
    print("c ",end = '\n\n')

    for k in tableau:
        for eq in k:
            print(Fraction(str(eq)).limit_denominator(100), end ='\t')
        print('\n')
```

```
# Function to check for optimality
def is_optimal(tableau):
    # Optimality is reached only when z < 0
    z = tableau[-1]
    # z[-1] always equals 0
    # any() would return a boolean value
    return not any(i > 0 for i in z[:-1])

# Function to find pivot element
def find_pivot_element(tableau):
    '''
    pivot column (pc) is the maximum value of z
    pivot row (pr) is the min of ratios of b values with pivot column values
    pr = index(min( [ b[i] / pc[i] ]))
    '''
    z = tableau[-1]
    ipc = next(i for i, x in enumerate(z[:-1]) if x > 0)
    ratio = []
    for eq in tableau[:-1]:
        el = eq[ipc]
        ratio.append(math.inf if el < 0 else eq[-1] / el)
    ipr = ratio.index(min(ratio))
    return ipr,ipc


def pivot_step(tableau,pivot_position):
    # finding zj
    new_tableau = [[] for i in tableau]
    ipr,ipc = pivot_position
    pivot_element = tableau[ipr][ipc]
    # Dividing the pivot row by pivot element
    new_tableau[ipr] = [el/pivot_element for el in tableau[ipr]]
    # Changing the other rows
    for eq_i,eq in enumerate(tableau):
        if eq_i != ipr:
            new_tableau[eq_i] = [
                tableau[eq_i][i] - new_tableau[ipr][i] * tableau[eq_i][ipc]
                for i in range(len(new_tableau[ipr]))
                ]
    return new_tableau
```

```
def simplex(z,A,b):
    # Converting the equations to tableau
    tableau = to_tableau(z,A,b)
    print("initial tableau\n\n")
    print_table(tableau)
    find_pivot_element(tableau)
    it = 1
    while not is_optimal(tableau):
        pivot_position = find_pivot_element(tableau)
        tableau = pivot_step(tableau,pivot_position)
        print('\n\niteration: {0}\n'.format(it))
        print_table(tableau)
        it+=1
    return tableau



def is_basic(column):
    return sum(column) == 1 and len([c for c in column if c == 0]) == len(column) - 1

def get_solution(tableau):
    columns = np.array(tableau).T
    solutions = []
    for column in columns[:-1]:
        solution = 0
        if is_basic(column):
            one_index = column.tolist().index(1)
            solution = columns[-1][one_index]
        solutions.append(solution)
    return solutions

def show_solutions(tableau):
    solutions = get_solution(tableau)
    var = ["x{0}".format(i+1) for i in range(len(tableau)-1)]
    for s,v in zip(solutions,var):
        print("{0} : {1}".format(v,Fraction(str(s)).limit_denominator(100)),end = '\n')
    print("z : {0}".format(Fraction(str(tableau[-1][-1])).limit_denominator(100)))

def main():

    '''
    Consider the LPP

    maximize z = 12x1 + 16x2
    ST:
    1) 10x1 + 20x2 + x3 = 120
    2)  8x1 +  8x2 + x4 = 80
    3)  x1,x2,x3,x4 > 0
    '''
    z = [2,-3,6,0,0,0]
    A = [
            [ 3,-1, 2, 1, 0, 0],
            [-2,-4, 0, 0, 1, 0],
            [-4, 3, 8, 0, 0, 1]
        ]
    b = [ 7,12,10]
    show_solutions(simplex(z,A,b))

if __name__ == "__main__":
    main()

from simplex import *

import numpy as np

z = np.array([5,7])
A = np.array([[-2,3],[6,1]])
b = np.array([6,30])

# Finding the initial optimal solution by using simplex method

tableau = simplex(z,A,b)
```

SAMPLE INPUT-OUTPUT

GENETIC ALGORITHM

AIM

To explore and implement genetic algorithms as a metaheuristic optimization technique, and to use evolutionary principles to find approximate solutions to complex optimization problems in various domains.

PROGRAM

```
#Genetic Algorithm

import random
import numpy as np
import math
import matplotlib.pyplot as plt



# Function to convert a binary list to a single integer
def binary_to_int(binary_list):
    decimal_value = 0
    power = len(binary_list) - 1

    for digit in binary_list:
        decimal_value += digit * (2 ** power)
        power -= 1

    return decimal_value

def plot_fitness(function,discrete_x,discrete_y,lim_range,generation):
    # Create x values for the continuous function
    # Creating 100 values in the lim_range of 0 to 16
    x = np.linspace(0, lim_range+1, 100)
    # Calculate the y values for the continuous function
    y_continuous = function(x)

    # Create the plot
    plt.plot(x, y_continuous, label='optimizing function at generation {0}'.format
    (generation))
    sc = plt.scatter(discrete_x, discrete_y, color='red', label='Population Values')

    # Add labels, title, and legend
    plt.xlabel('x-axis')
    plt.ylabel('y-axis')
    plt.title('Optimizing function and population values')
    plt.legend()

    # Display the plot
    plt.grid(True)
    return sc

def update(sc,discrete_x,discrete_y):
    print("discrete x : ",discrete_x)
    plt.pause(0.01)
    sc.set_offsets(np.c_[discrete_x,discrete_y])
    plt.draw()

# Randomly initialize a population
def init_population(population_size, bits):
    population = []
    for _ in range(population_size):
        chromosome = [random.randint(0,1) for _ in range(bits)]
        population.append(chromosome)
    return population

# Calculate the fitness for the entire population
def get_fitness(population,fitness_fn):
    fitness = []
    int_values = []
    for chr in population:
        # Convert the list into an integer
        int_conv = binary_to_int(chr)
        int_values.append(int_conv)
        fitness.append(fitness_fn(int_conv))

    return fitness,int_values

# Select a randon individual based o roulette wheel selection
def roulette_wheel_selection(fitness):
    # Determining the probabilities of selection
    fit_sum = sum(fitness)
    fitness_prob_weights = [1 - (fit/fit_sum )for fit in fitness]
    roulette_wheel = []
    roulette_wheel.append(fitness_prob_weights[0])

    # Generating a roulette wheel
    for fwi,fw in enumerate(fitness_prob_weights):
        if fwi != 0:
            roulette_wheel.append(roulette_wheel[fwi-1]+fw)
    print("roulette wheel weights : ",roulette_wheel)

    # Generate a random number
    rand_num = random.random()
    print("random number : ",rand_num)
    print("roulette wheel : ",roulette_wheel)
    for i,weight in enumerate(roulette_wheel):
        if rand_num  < weight:
            print("selected : ",i)
            return i

# Function to perform single point crossover in 2 inds
def single_point_crossover(parent1,parent2):
    # Ensure both parents have the same length
    assert len(parent1) == len(parent2)
    # Choosing a random poin to swap from
    x = random.randint(0,len(parent1))
    # Randomly select a crossover point
    crossover_point = random.randint(1, len(parent1) - 1)

    # Perform crossover to create two offspring individuals
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]

    return offspring1, offspring2

# Defining bitflip mutation
def bitflip_mutation(chromosome,mutation_rate):
    for i,chrome in enumerate(chromosome):
        # Generate a random number
        random_mutation_num = random.random()
        if random_mutation_num < mutation_rate:
            if chrome == 0:
                chromosome[i] = 1
            else:
                chromosome[i] = 0
    return chromosome




def genetic_optimize(fitness_fn,population_size,crossover_rate,mutation_rate,
lim_range,total_generations):
    plt.show()
    # Finding the number of bits required to initialize the population
    bits = int(math.log2(lim_range)+1)
    # Generating a random popultation
    population = init_population(population_size,bits)
    print("Initial population : ",population)
    fitness,int_values = get_fitness(population,fitness_fn)
    # Plotting the fitness function and fitness of the population
    plt_sc = plot_fitness(fitness_fn,int_values,fitness,lim_range,0)

    # Running the genetic algorithm for a set of generations
    for generation_i in range(total_generations):

        new_generation = []

        while len(new_generation) < population_size:
            # selecting 2 random individuals for crossover
            ind1 = roulette_wheel_selection(fitness)
            ind2 = roulette_wheel_selection(fitness)

            # Setting a random probability for genetic crossover
            crossover_prob = random.random()
            if crossover_prob < crossover_rate:
                offspring1, offspring2 = single_point_crossover
                (population[ind1], population[ind2])

                # Random mutation
                mutant_1 = bitflip_mutation(offspring1,mutation_rate)
                mutant_2 = bitflip_mutation(offspring2,mutation_rate)

                new_generation.append(mutant_1)
                new_generation.append(mutant_2)

        population = new_generation.copy()
        fitness,int_values = get_fitness(population,fitness_fn)

        # Plotting the fitness function and fitness of the population
        update(plt_sc,int_values,fitness)

        for i in fitness:
            print(i, end = " ")

    return max(fitness),population[fitness.index(max(fitness))]


def main():
    POPULATION_SIZE = 10
    CROSSOVER_RATE = 0.7
    MUTATION_RATE = 0.01
    FITNESS_FN = lambda x : -1*(x-9)**2
    LIM_RANGE = 127

    TOTAL_GENERATIONS = 50

    max_value, x_value = genetic_optimize(FITNESS_FN,POPULATION_SIZE,CROSSOVER_RATE,
    MUTATION_RATE,LIM_RANGE,TOTAL_GENERATIONS)
    print("Max value by genetic = {0} for x = {1} ".format(max_value,x_value))

main()
```

SAMPLE INPUT-OUTPUT

SWARM INTELLIGENCE ALGORITHM

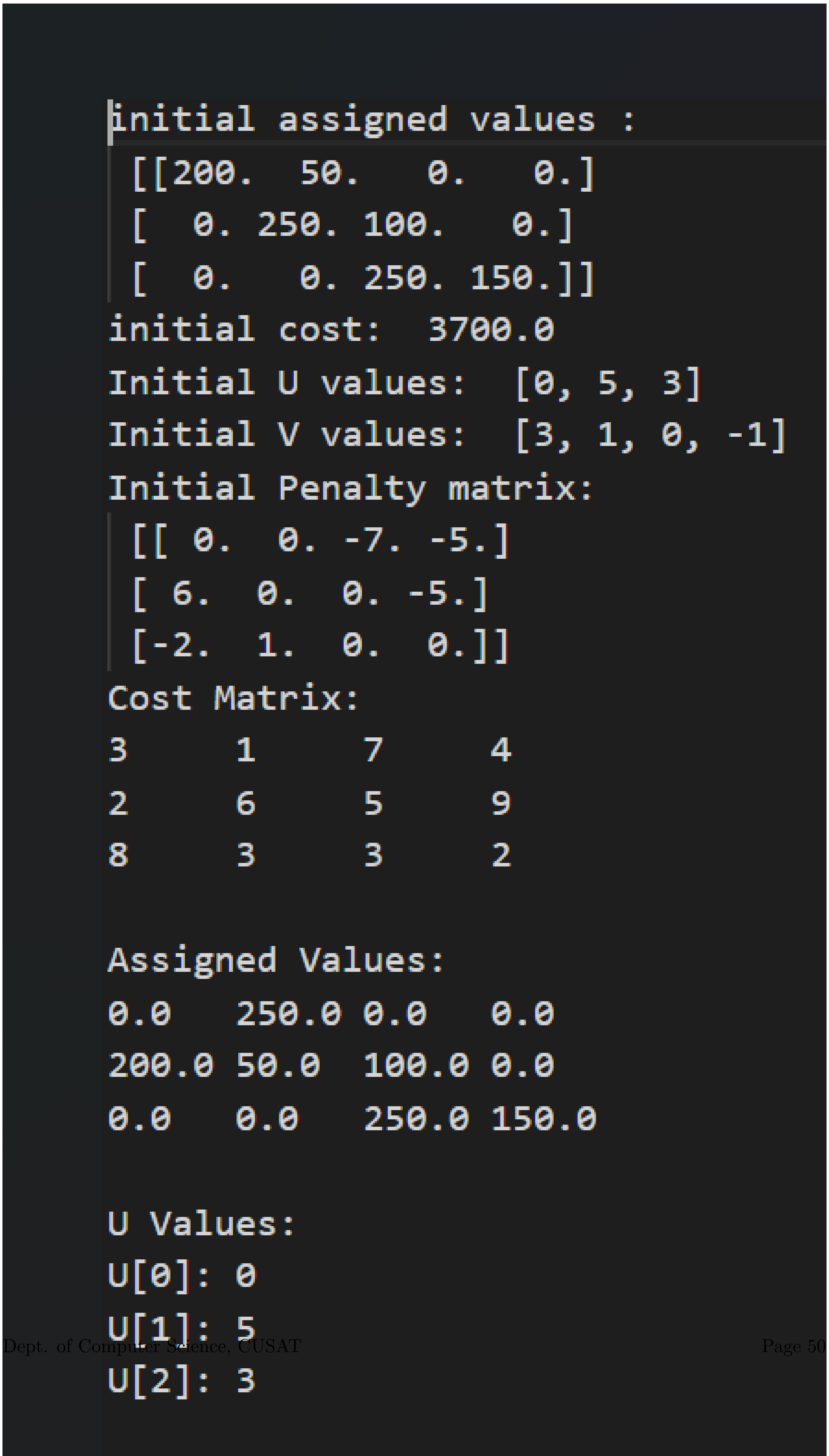
AIM

To use swarm intelligence algorithms for optimization tasks and implement them in Python.

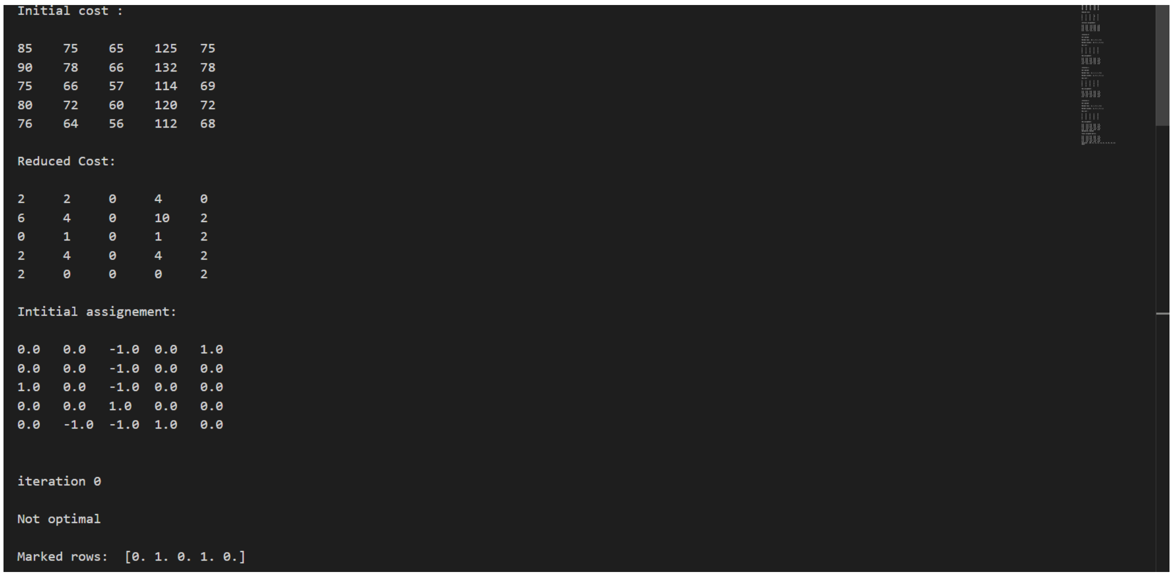
PROGRAM

```
#Swarm Intelligence

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
sns.set_context("notebook")

def opt_func(value):
    """The mathematical function to optimize. Here it calculates the distance to origin,
    i.e. optimal solution(minimum) is at 0.

    Arguments:
        value {np.ndarray} -- An individual value or frog

    Returns:
        float -- The output value or fitness of the frog
    """

    output = np.sqrt((value ** 2).sum())
    return output

def gen_frogs(frogs, dimension, sigma, mu):
    """Generates a random frog population from gaussian normal distribution.

    Arguments:
        frogs {int} -- Number of frogs
        dimension {int} -- Dimension of frogs/ Number of features
        sigma {int/float} -- Sigma of gaussian distribution
        mu {int/float} -- Mu of Gaussian distribution

    Returns:
        numpy.ndarray -- A frogs x dimension array
    """
    frogs = sigma * (np.random.rand(frogs, dimension)) + mu
    return frogs

def sort_frogs(frogs, mplx_no, opt_func):
    """Sorts the frogs in decending order of fitness by the given function.

    Arguments:
        frogs {numpy.ndarray} -- Frogs to be sorted
        mplx_no {int} -- Number of memeplexes, when divides frog number should return an
        integer otherwise frogs will be skipped
        opt_func {function} -- Function to determine fitness

    Returns:
        numpy.ndarray -- A memeplexes x frogs/memeplexes array of indices,
        [0, 0] will be the greatest frog
    """

    # Find fitness of each frog
    fitness = np.array(list(map(opt_func, frogs)))
    # Sort the indices in decending order by fitness
    sorted_fitness = np.argsort(fitness)
    # Empty holder for memeplexes
    memeplexes = np.zeros((mplx_no, int(frogs.shape[0]/mplx_no)))
    # Sort into memeplexes
    for j in range(memeplexes.shape[1]):
        for i in range(mplx_no):
            memeplexes[i, j] = sorted_fitness[i+(mplx_no*j)]
    return memeplexes

def local_search(frogs, memeplex, opt_func, sigma, mu):
    """Performs the local search for a memeplex.

    Arguments:
        frogs {numpy.ndarray} -- All the frogs
        memeplex {numpy.ndarray} -- One memeplex
        opt_func {function} -- The function to optimize
        sigma {int/float} -- Sigma for the gaussian distribution by
        which the frogs were created
        mu {int/float} -- Mu for the gaussian distribution by which
        the frogs were created
    Returns:
        numpy.ndarray -- The updated frogs, same dimensions
    """

    # Select worst, best, greatest frogs
    frog_w = frogs[int(memeplex[-1])]
    frog_b = frogs[int(memeplex[0])]
    frog_g = frogs[0]
    # Move worst wrt best frog
    frog_w_new = frog_w + (np.random.rand() * (frog_b - frog_w))
    # If change not better, move worst wrt greatest frog
    if opt_func(frog_w_new) > opt_func(frog_w):
        frog_w_new = frog_w + (np.random.rand() * (frog_g - frog_w))
    # If change not better, random new worst frog
    if opt_func(frog_w_new) > opt_func(frog_w):
        frog_w_new = gen_frogs(1, frogs.shape[1], sigma, mu)[0]
    # Replace worst frog
    frogs[int(memeplex[-1])] = frog_w_new
    return frogs

def shuffle_memeplexes(frogs, memeplexes):
    """Shuffles the memeplexes without sorting them.

    Arguments:
        frogs {numpy.ndarray} -- All the frogs
        memeplexes {numpy.ndarray} -- The memeplexes

    Returns:
        numpy.ndarray -- A shuffled memeplex, unsorted, same dimensions
    """

    # Flatten the array
    temp = memeplexes.flatten()
    #Shuffle the array
    np.random.shuffle(temp)
    # Reshape
    temp = temp.reshape((memeplexes.shape[0], memeplexes.shape[1]))
    return temp

def sfla(opt_func, frogs=30, dimension=2, sigma=1, mu=0, mplx_no=5,
mplx_iters=10, solun_iters=50):
    """Performs the Shuffled Leaping Frog Algorithm.

    Arguments:
        opt_func {function} -- The function to optimize.

    Keyword Arguments:
        frogs {int} -- The number of frogs to use (default: {30})
        dimension {int} -- The dimension/number of features (default: {2})
        sigma {int/float} -- Sigma for the gaussian normal distribution to
        create the frogs (default: {1})
        mu {int/float} -- Mu for the gaussian normal distribution to
        create the frogs (default: {0})
        mplx_no {int} -- Number of memeplexes, when divides frog number should return
        an integer otherwise frogs will be skipped (default: {5})
        mplx_iters {int} -- Number of times a single memeplex will
        be iterated before shuffling (default: {10})
        solun_iters {int} -- Number of times the memeplexes will
        be shuffled (default: {50})

    Returns:
        tuple(numpy.ndarray, numpy.ndarray, numpy.ndarray) -- [description]
    """

    # Generate frogs around the solution
    frogs = gen_frogs(frogs, dimension, sigma, mu)
    # Arrange frogs and sort into memeplexes
    memeplexes = sort_frogs(frogs, mplx_no, opt_func)
    # Best solution as greatest frog
    best_solun = frogs[int(memeplexes[0, 0])]
    # For the number of iterations
    for i in range(solun_iters):
        # Shuffle memeplexes
        memeplexes = shuffle_memeplexes(frogs, memeplexes)
        # For each memeplex
        for mplx_idx, memeplex in enumerate(memeplexes):
            # For number of memeplex iterations
            for j in range(mplx_iters):
                # Perform local search
                frogs = local_search(frogs, memeplex, opt_func, sigma, mu)
            # Rearrange memeplexes
            memeplexes = sort_frogs(frogs, mplx_no, opt_func)
            # Check and select new best frog as the greatest frog
            new_best_solun = frogs[int(memeplexes[0, 0])]
            if opt_func(new_best_solun) < opt_func(best_solun):
                best_solun = new_best_solun
    return best_solun, frogs, memeplexes.astype(int)

def main():
    # Run algorithm
    solun, frogs, memeplexes = sfla(opt_func, 12, 2, 1, 0, 3, 2, 1)
    print("Optimal Solution (closest to zero): {}".format(solun))
    # Place memeplexes
    for idx, memeplex in enumerate(memeplexes):
        plt.scatter(frogs[memeplex, 0], frogs[memeplex, 1], marker='x',
        label="memeplex {}".format(idx))
    plt.scatter(solun[0], solun[1], marker='o', label="Optimal Solution")
    plt.scatter(0, 0, marker='*', label='Actual Solution')
    # Plot properties
    plt.legend()
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Shuffled Frogs")
    # Show plot
    plt.show()

if __name__ == '__main__':
    main()
```

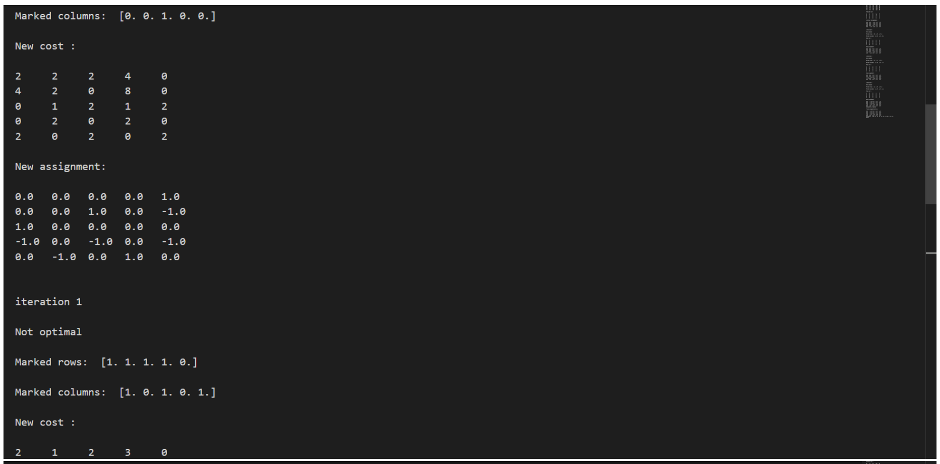
SAMPLE INPUT-OUTPUT



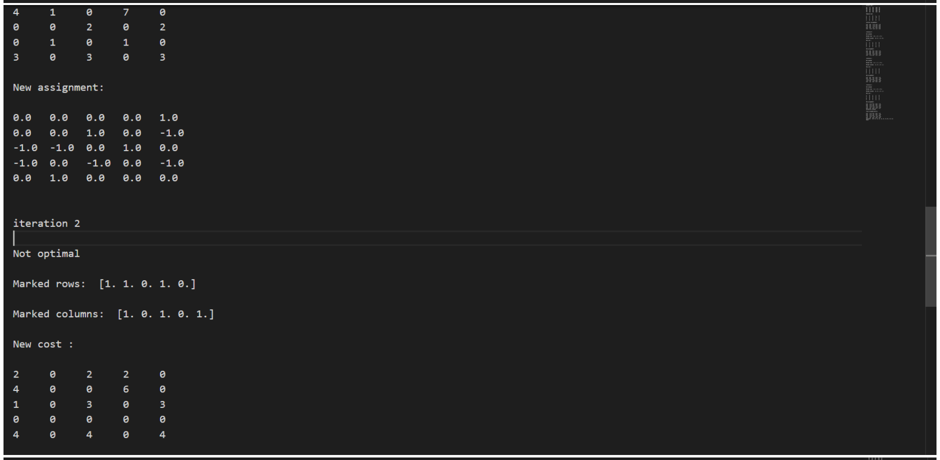
```
initial assigned values :
 [[200.  50.   0.   0.]
 [  0. 250. 100.   0.]
 [  0.   0. 250. 150.]]
initial cost:  3700.0
Initial U values:  [0, 5, 3]
Initial V values:  [3, 1, 0, -1]
Initial Penalty matrix:
 [[ 0.  0. -7. -5.]
 [ 6.  0.  0. -5.]
 [-2.  1.  0.  0.]]
Cost Matrix:
3     1     7     4
2     6     5     9
8     3     3     2

Assigned Values:
0.0   250.0 0.0   0.0
200.0 50.0  100.0 0.0
0.0   0.0   250.0 150.0

U Values:
U[0]: 0
U[1]: 5
U[2]: 3
```



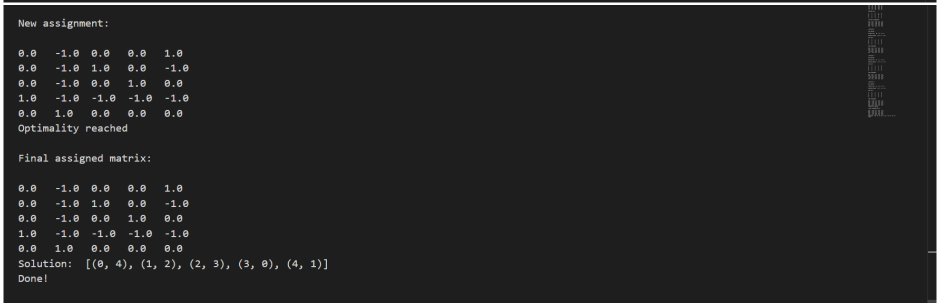
```
Initial cost :

85    75    65    125   75
90    78    66    132   78
75    66    57    114   69
80    72    60    120   72
76    64    56    112   68

Reduced Cost:

2     2     0     4     0
6     4     0     10    2
0     1     0     1     2
2     4     0     4     2
2     0     0     0     2

Intitial assignement:

0.0   0.0   -1.0  0.0   1.0
0.0   0.0   -1.0  0.0   0.0
1.0   0.0   -1.0  0.0   0.0
0.0   0.0   1.0   0.0   0.0
0.0   -1.0  -1.0  1.0   0.0


iteration 0

Not optimal

Marked rows:  [0. 1. 0. 1. 0.]
```



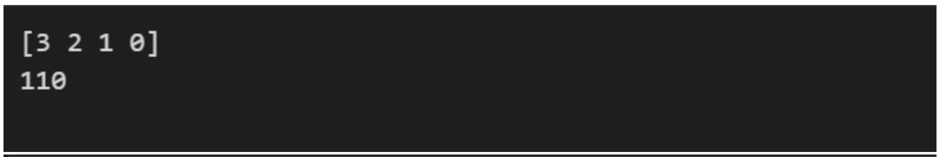
```
Marked columns:  [0. 0. 1. 0. 0.]

New cost :

2     2     2     4     0
4     2     0     8     0
0     1     2     1     2
0     2     0     2     0
2     0     2     0     2

New assignment:

0.0   0.0   0.0   0.0   1.0
0.0   0.0   1.0   0.0   -1.0
1.0   0.0   0.0   0.0   0.0
-1.0  0.0   -1.0  0.0   -1.0
0.0   -1.0  0.0   1.0   0.0


iteration 1

Not optimal

Marked rows:  [1. 1. 1. 1. 0.]

Marked columns:  [1. 0. 1. 0. 1.]

New cost :

2     1     2     3     0
```



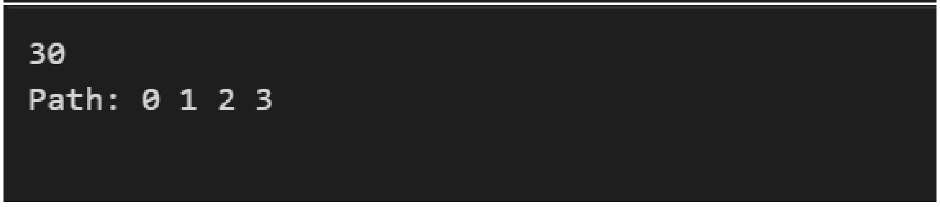
```
4     1     0     7     0
0     0     2     0     2
0     1     0     1     0
3     0     3     0     3

New assignment:

0.0   0.0   0.0   0.0   1.0
0.0   0.0   1.0   0.0   -1.0
-1.0  -1.0  0.0   1.0   0.0
-1.0  0.0   -1.0  0.0   -1.0
0.0   1.0   0.0   0.0   0.0


iteration 2

Not optimal

Marked rows:  [1. 1. 0. 1. 0.]

Marked columns:  [1. 0. 1. 0. 1.]

New cost :

2     0     2     2     0
4     0     0     6     0
1     0     3     0     3
0     0     0     0     0
4     0     4     0     4
```



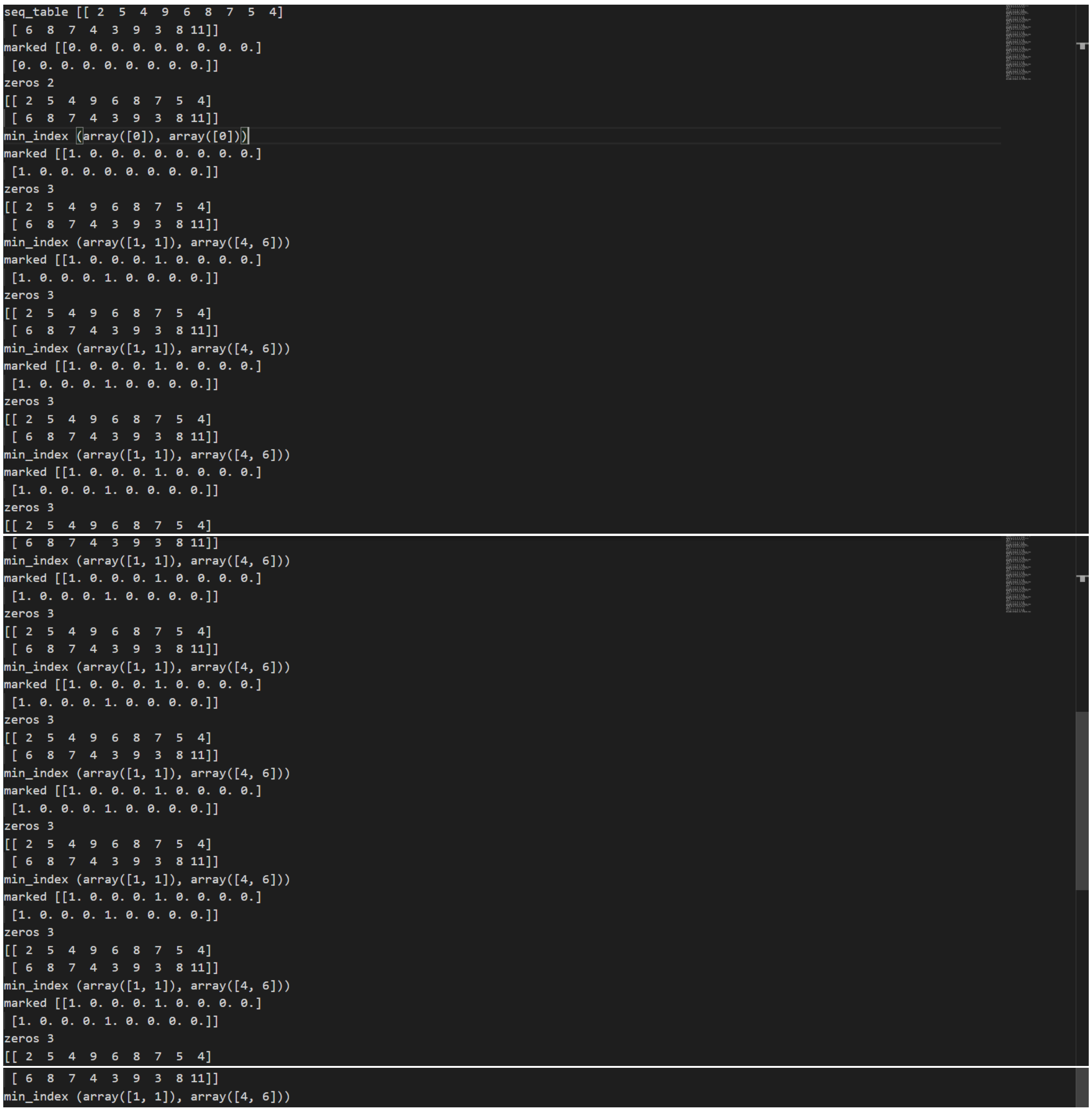
```
New assignment:

0.0   -1.0  0.0   0.0   1.0
0.0   -1.0  1.0   0.0   -1.0
0.0   -1.0  0.0   1.0   0.0
1.0   -1.0  -1.0  -1.0  -1.0
0.0   1.0   0.0   0.0   0.0
Optimality reached

Final assigned matrix:

0.0   -1.0  0.0   0.0   1.0
0.0   -1.0  1.0   0.0   -1.0
0.0   -1.0  0.0   1.0   0.0
1.0   -1.0  -1.0  -1.0  -1.0
0.0   1.0   0.0   0.0   0.0
Solution:  [(0, 4), (1, 2), (2, 3), (3, 0), (4, 1)]
Done!
```



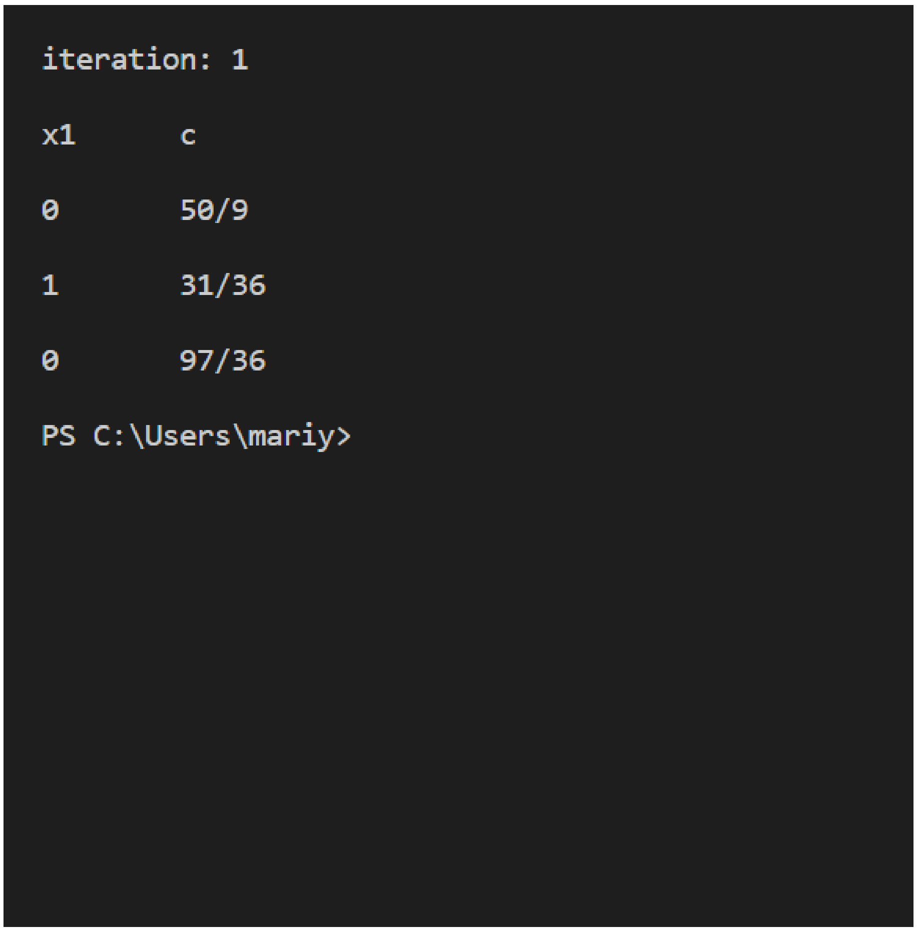
```
[3 2 1 0]
110
```



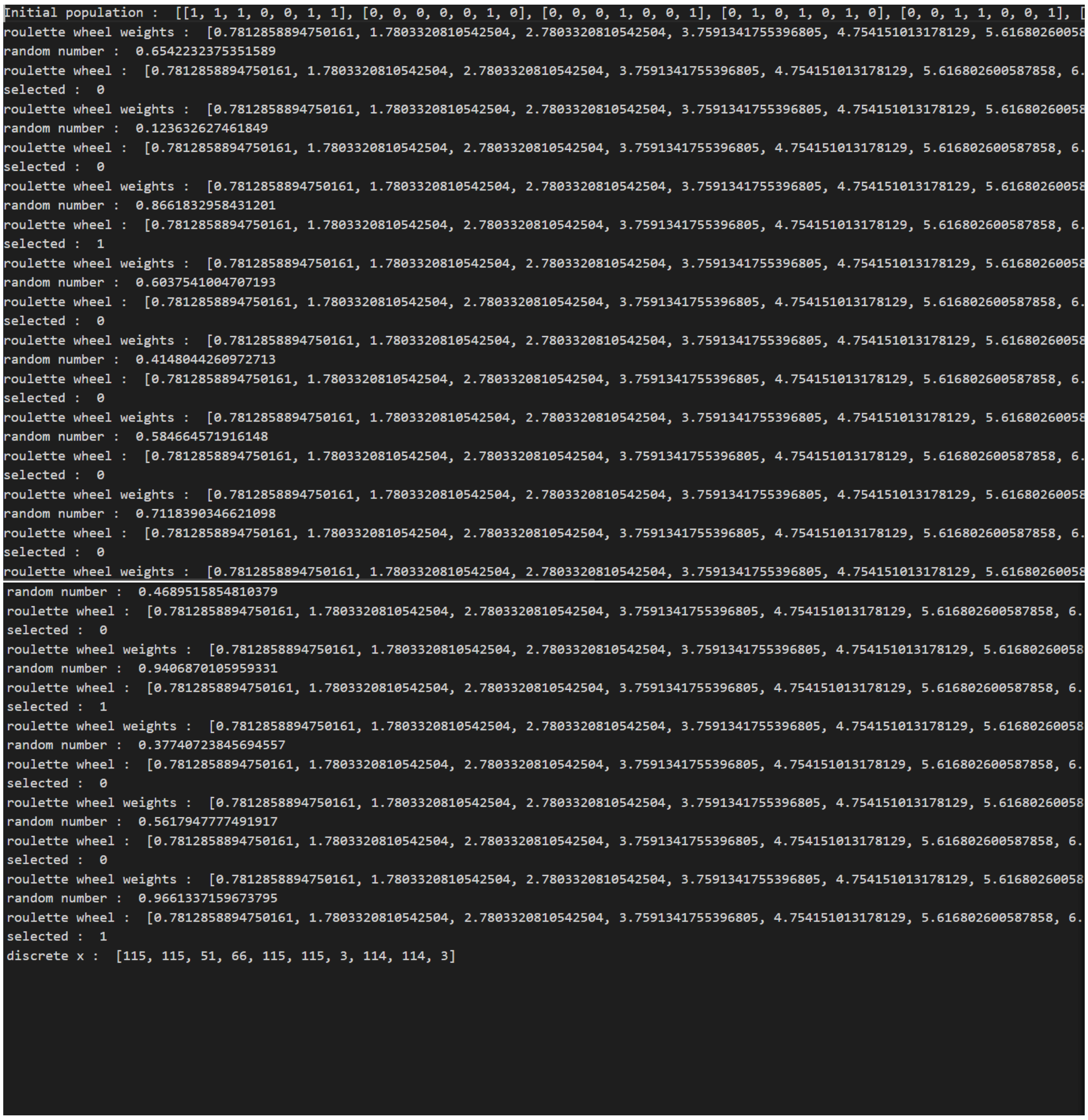
```
30
Path: 0 1 2 3
```



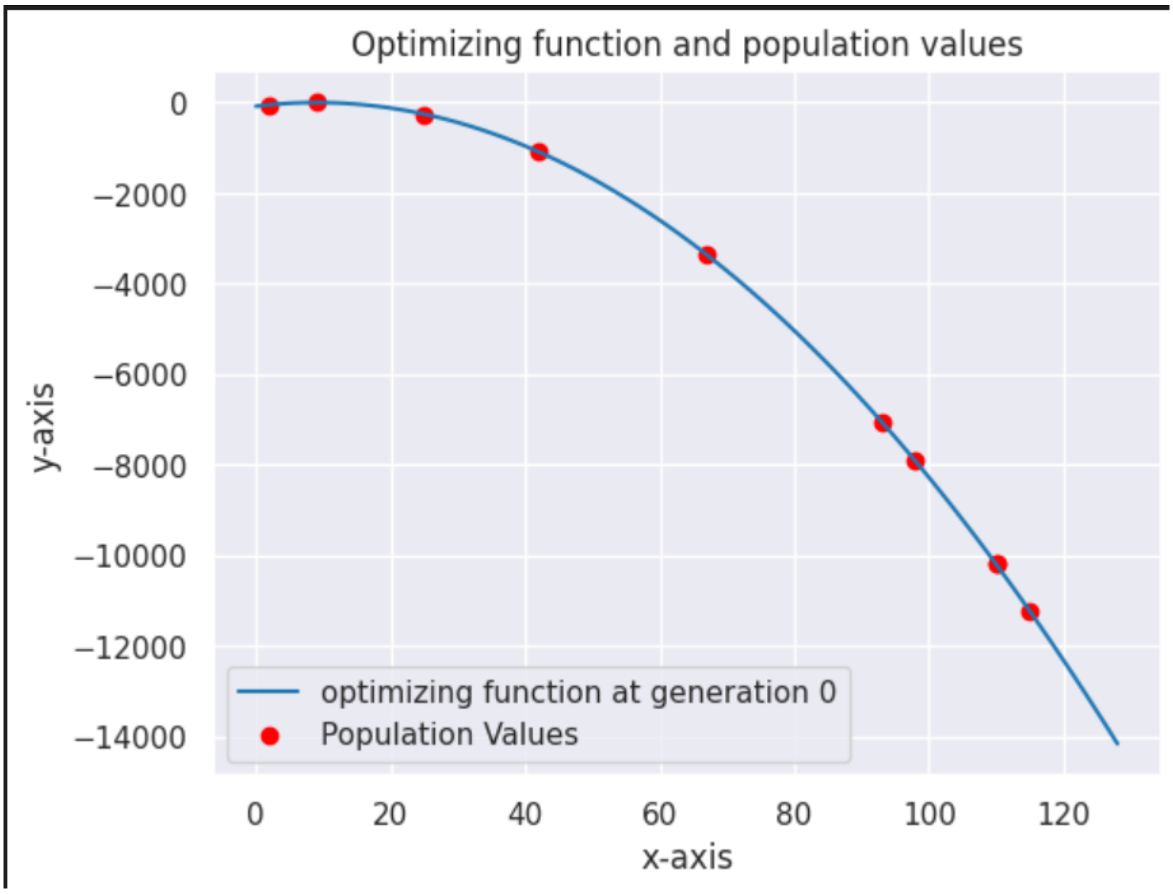
```
seq_table [[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
marked [[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
zeros 2
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([0]), array([0]))
marked [[1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
marked [[1. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 0.]]
zeros 3
[[ 2  5  4  9  6  8  7  5  4]
 [ 6  8  7  4  3  9  3  8 11]]
min_index (array([1, 1]), array([4, 6]))
```

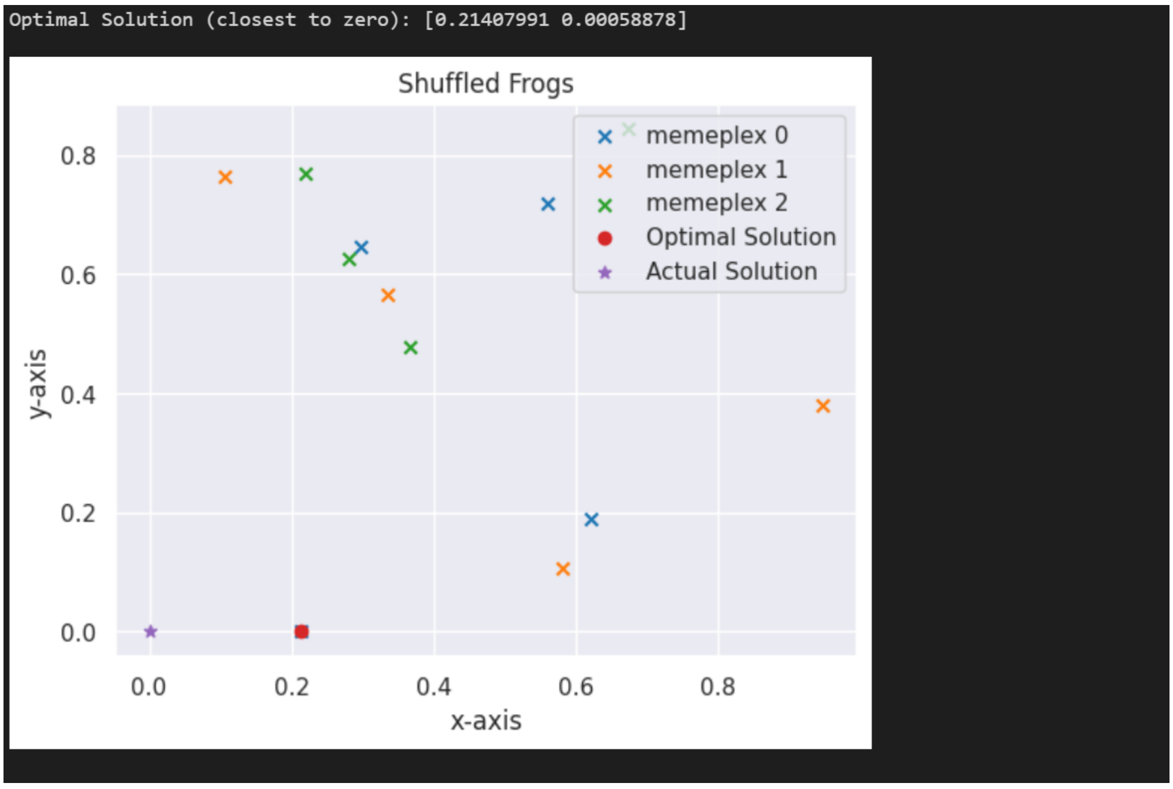


```
iteration: 1

x1      c

0       50/9

1       31/36

0       97/36

PS C:\Users\mariy>
```



```
Initial population :  [[1, 1, 1, 0, 0, 1, 1], [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 1, 0, 0, 1], [0, 1, 0, 1, 0, 1, 0], [0, 0, 1, 1, 0, 0, 1], [
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.6542232375351589
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.123632627461849
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.8661832958431201
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  1
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.6037541004707193
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.4148044260972713
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.584664571916148
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.7118390346621098
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.4689515854810379
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.9406870105959331
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  1
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.37740723845694557
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.5617947777491917
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  0
roulette wheel weights :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.61680260058
random number :  0.9661337159673795
roulette wheel :  [0.7812858894750161, 1.7803320810542504, 2.7803320810542504, 3.7591341755396805, 4.754151013178129, 5.616802600587858, 6.
selected :  1
discrete x :  [115, 115, 51, 66, 115, 115, 3, 114, 114, 3]
```





```
Optimal Solution (closest to zero): [0.21407991 0.00058878]
```