**PYCHARM ALGORITHMS**

**Algorithms for finding the optimal solution**

**Particular case:**

import numpy as np

import matplotlib . pyplot as plt

from scipy . optimize import linprog

# Define the coefficients of the objective

function

c = [ -3 , -2] # Example : Maximize 3 x + 2 y

# Define the inequality constraints ( Ax <= b

)

A = [[1 , 2] , [2 , 1] , [1 , -1]] # Example

Corner Points Coordinates Value of Z = 5x1 + 7x2

O (0,0) 0

A (3.5,0) 17.5

B (1.6,2.4) 24.1

C (1.6,2.4) 24.8

D (0,3) 21

constraints

b = [10 , 8 , 3]

# Define the range for the variables

x = np . linspace (0 , 10 , 400)

y = np . linspace (0 , 10 , 400)

# Create a meshgrid for plotting

X , Y = np . meshgrid (x , y )

# Plot the constraints

plt . figure ( figsize =(8 , 8) )

for i in range ( len ( A ) ) :

plt . plot (x , ( b [ i ] - A [ i ][0] \* x ) / A [ i

][1] ,

slabel = f ’ Constraint { i +1} ’)

# Plot the feasible region

plt . fill\_between (x , np . minimum (( b [0] - A

[0][0] \*

x ) / A [0][1] , ( b [1] - A [1][0] \* x ) / A

[1][1]) ,

where =( x >= 0) , color = ’ gray ’ , alpha =0.5)

# Plot the objective function lines

for z in range (0 , 30 , 5) :

plt . plot (x , ( z - c [0] \* x ) / c [1] ,

linestyle = ’ -- ’ ,

label = f ’ Objective function ( z ={ z }) ’)

# Solve the linear programming problem

res = linprog (c , A\_ub =A , b\_ub =b , method = ’

highs ’)

# Plot the optimal solution

plt . plot ( res . x [0] , res . x [1] , ’ ro ’ ,

label = ’ Optimal Solution ’)

# Set plot limits and labels

plt . xlim (0 , 10)

plt . ylim (0 , 10)

plt . xlabel ( ’x ’)

plt . ylabel ( ’y ’)

plt . title ( ’ Graphical Method for Linear

Programming ’)

plt . legend ()

plt . grid ( True )

plt . show ()

# Print the rsults

print ( ’ Optimal value : ’ , - res . fun )

print ( ’ Optimal solution : ’ , res . x )

**Output:**

Optimal value: 14.0

Optimal solution: 2.0, 4.0.

**General case:**

import numpy as np

import matplotlib . pyplot as plt

from itertools import combinations

def find\_intersection (A , b ) :

"""

Finds all intersection points of the constraints to

determine corner points .

A : Coefficients of inequality constraints .

b : Right - hand side values of the constraints .

Returns :

Corner points as a list .

"""

corner\_points = []

for (i , j ) in combinations ( range ( len ( A ) ) , 2) : #

Pairwise combination of constraints

A\_sub = np . array ([ A [ i ] , A [ j ]])

b\_sub = np . array ([ b [ i ] , b [ j ]])

try :

point = np . linalg . solve ( A\_sub , b\_sub ) #

Solves Ax = b

corner\_points . append ( point )

except np . linalg . LinAlgError :

pass # Skip if the constraints are parallel

( no intersection )

return corner\_points

def plot\_two\_point\_feasible\_region (c , A , b , xlim , ylim ,

problem\_type = ’ max ’) :

"""

Solves and plots an LPP with a feasible region

outlined using dotted lines for two points .

"""

if problem\_type == ’ max ’:

c = [ - coef for coef in c ]

x = np . linspace ( xlim [0] , xlim [1] , 400)

plt . figure ( figsize =(10 , 10) )

# Plot each constraint

for i , row in enumerate ( A ) :

y = ( b [ i ] - row [0] \* x ) / row [1]

plt . plot (x , y , label = f ’ Constraint { i + 1} ’)

# Find intersection points and validate feasibility

corner\_points = find\_intersection (A , b )

feasible\_points = []

for point in corner\_points :

if all ( np . dot ( row , point ) <= bound for row ,

bound in zip (A , b ) ) :

feasible\_points . append ( point )

# Ensure we have exactly two points

if len ( feasible\_points ) == 2:

# Convert feasible points to NumPy array

feasible\_points = np . array ( feasible\_points )

# Draw the dotted - line boundary between the two

points

plt . plot ( feasible\_points [: , 0] , feasible\_points

[: , 1] , ’k - - ’ , label = ’ Feasible Region

Boundary ’)

elif len ( feasible\_points ) < 2:

print ( " Warning : Not enough feasible points to

form a two - point feasible region . " )

else :

print ( " Warning : Feasible region has more than

two points , unexpected for this case . " )

# Evaluate objective function and find the optimal

solution

optimal\_value = None

optimal\_point = None

for point in feasible\_points :

value = np . dot (c , point )

if optimal\_value is None or value >

optimal\_value :

optimal\_value = value

optimal\_point = point

# Plot feasible corner points

for point in feasible\_points :

plt . plot ( point [0] , point [1] , ’ go ’ , label = f "

Feasible Point ({ point [0]:.2 f } , { point [1]:.2 f

}) " )

# Plot optimal solution

if optimal\_point is not None :

plt . plot ( optimal\_point [0] , optimal\_point [1] , ’ ro

’ , label = f " Optimal Solution ({ optimal\_point

[0]:.2 f } , { optimal\_point [1]:.2 f }) " )

print ( f " Optimal solution : { optimal\_point } ,

Optimal value : { optimal\_value } " )

# Set plot details

plt . xlim ( xlim )

plt . ylim ( ylim )

plt . xlabel ( ’x ’)

plt . ylabel ( ’y ’)

plt . title ( ’ Linear Programming : Feasible Region for

Two Points ’)

plt . legend ()

plt . grid ( True )

plt . show ()

# User Input Section

print ( " Solve Linear Programming Problem with Feasible

Region for Two Points " )

c = list ( map ( float , input ( " Enter the coefficients for

the objective function ( e . g . , -3 -2) : " ) . split () ) )

num\_constraints = int ( input ( " Enter the number of

constraints : " ) )

A = []

b = []

for i in range ( num\_constraints ) :

constraint = list ( map ( float , input ( f " Enter the

coefficients for constraint { i + 1} ( e . g . , 1 2) :

" ) . split () ) )

A . append ( constraint )

b\_value = float ( input ( f " Enter the right - hand side

value for constraint { i + 1}: " ) )

b . append ( b\_value )

xlim = tuple ( map ( float , input ( " Enter the x - axis limits (

e . g . , 0 10) : " ) . split () ) )

ylim = tuple ( map ( float , input ( " Enter the y - axis limits (

e . g . , 0 10) : " ) . split () ) )

problem\_type = input ( " Enter the problem type ( ’ max ’ for

maximization , ’ min ’ for minimization ) : " ) . strip () .

lower ()

# Call Function

p lot\_two\_point\_feasible\_region (c , A , b , xlim , ylim ,

problem\_type )

**Output:**

Enter the coefficients for the objective function (e.g., -3 -2): -6 -4

Enter the number of constraints: 3 Enter the coefficients for constraint 1

(e.g., 1 2): -2 1

Enter the right-hand side value for constraint 1: 2

Enter the coefficients for constraint 2 (e.g., 1 2): 1 -1

Enter the right-hand side value for constraint 2: 2

Enter the coefficients for constraint 3 (e.g., 1 2): 3 2

Enter the right-hand side value for constraint 3: 9

Enter the x-axis limits (e.g., 0 10): 0 10

Enter the y-axis limits (e.g., 0 10): 0 10

Enter the problem type (’max’ for maximization, ’min’ for minimization):

max

Optimal solution: [2.6 0.6], Optimal value: 18.0

**Python Code for Assignment Problem**

**Algorithm for particular case:**

import numpy as np

from scipy . optimize import linear\_sum\_assignment

def assignment\_problem ( cost\_matrix ) :

"""

Solves the Assignment Problem using the

Hungarian

Algorithm .

Args :

cost\_matrix ( list of lists ) : Cost matrix

where

cost\_matrix [ i ][ j ] represents the cost of

assigning task

j to agent i .

Returns :

row\_ind ( list ) : List of row indices of

the optimal assignment .

col\_ind ( list ) : List of column indicesof

the optimal assignment .

total\_cost ( float ) : Total cost of

the optimal assignment .

"""

# Convert the cost matrix to a numpy array

cost\_matrix = np . array ( cost\_matrix )

# Solve the assignment problem using the

Hungarian

algorithm

row\_ind , col\_ind = linear\_sum\_assignment (

cost\_matrix )

# Calculate the total cost

total\_cost = cost\_matrix [ row\_ind , col\_ind ]. sum ()

return row\_ind , col\_ind , total\_cost

def print\_table ( table , title ) :

print ( f " \ n { title }: " )

for row in table :

print ( row )

def main () :

print ( " Assignment Problem Solver " )

# Define the cost matrix

cost\_matrix =

[

[8 , 7 , 6] ,

[5 , 7 , 8] ,

[6 , 8 , 7]

]

# Print the initial cost matrix

print\_table ( cost\_matrix , " Initial Cost Matrix " )

# Solve the assignment problem

row\_ind , col\_ind , total\_cost =

assignment\_problem

( cost\_matrix )

# Create the assignment table

assignment\_table = np . zeros\_like ( cost\_matrix )

for i , j in zip ( row\_ind , col\_ind ) :

assignment\_table [i , j ] = 1

# Print the assignment table

print\_table ( assignment\_table , " Assignment Table "

)

# Print the results

print ( " \ nOptimal Assignment : " )

for i , j in zip ( row\_ind , col\_ind ) :

print ( f " Agent { i + 1} is assigned to

Task

{ j + 1} " )

print ( f " \ nTotal Cost : { total\_cost } " )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

\ end { minted }

\ noindent \ textbf { Output :} \\

\ noindent Initial Cost Matrix :

\[

\ begin { bmatrix }

8 & 7 & 6 \\

5 & 7 & 8 \\

6 & 8 & 7 \\

\ end { bmatrix }

\]

\ noindent Assignment Table : \\

\[

\ begin { bmatrix }

0 & 0 & 1 \\

1 & 0 & 0 \\

0 & 1 & 0 \\

\ end { bmatrix }

\]

\ noindent Optimal Assignment : \\

Agent 1 is assigned to Task 3 \\

Agent 2 is assigned to Task 1 \\

Agent 3 is assigned to Task 2 \\

\ noindent Thus , total Cost is 19.\\

\ noindent \ textbf { Algorithm for generalised case }

\ begin { minted }[ fontsize =\ small ]{ python }

import numpy as np

from scipy . optimize import linear\_sum\_assignment

def assignment\_problem ( cost\_matrix ) :

"""

Solves the Assignment Problem using the Hungarian

Algorithm .

Args :

cost\_matrix ( list of lists ) : Cost matrix where

cost\_matrix [ i ][ j ]

represents the cost of assigning task j to agent

i .

Returns :

row\_ind ( list ) : List of row indices of the

optimal

assignment .

col\_ind ( list ) : List of column indices of the

optimal

assignment .

total\_cost ( float ) : Total cost of the optimal

assignment .

"""

# Convert the cost matrix to a numpy array

cost\_matrix = np . array ( cost\_matrix )

# Solve the assignment problem using the Hungarian

algorithm

row\_ind , col\_ind = linear\_sum\_assignment ( cost\_matrix

)

# Calculate the total cost

total\_cost = cost\_matrix [ row\_ind , col\_ind ]. sum ()

return row\_ind , col\_ind , total\_cost

def main () :

print ( " Assignment Problem Solver " )

# Get the number of agents and tasks

num\_agents = int ( input ( " Enter the number of agents :

" ) )

num\_tasks = int ( input ( " Enter the number of tasks : " )

)

# Get the cost matrix

cost\_matrix = []

for i in range ( num\_agents ) :

row = []

for j in range ( num\_tasks ) :

cost = float ( input ( f " Enter the cost of

assigning

task { j +1} to agent s { i +1}: " ) )

row . append ( cost )

cost\_matrix . append ( row )

# Solve the assignment problem

row\_ind , col\_ind , total\_cost = assignment\_problem

( cost\_matrix )

# Print the results

print ( " \ nOptimal Assignment : " )

for i , j in zip ( row\_ind , col\_ind ) :

print ( f " Agent { i +1} is assigned to Task { j +1} " )

print ( f " \ nTotal Cost : { total\_cost } " )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

**Output:**

Assignment Problem Solver:

Enter the number of agents: 4

Enter the number of tasks: 4

Enter the cost of assigning task 1 to agent 1: 18

Enter the cost of assigning task 2 to agent 1: 26

Enter the cost of assigning task 3 to agent 1: 17

Enter the cost of assigning task 4 to agent 1: 11

Enter the cost of assigning task 1 to agent 2: 13

Enter the cost of assigning task 2 to agent 2: 28

Enter the cost of assigning task 3 to agent 2: 14

Enter the cost of assigning task 4 to agent 2: 26

Enter the cost of assigning task 1 to agent 3: 38

Enter the cost of assigning task 2 to agent 3: 19

Enter the cost of assigning task 3 to agent 3: 18

Enter the cost of assigning task 4 to agent 3: 15

Enter the cost of assigning task 1 to agent 4: 19

Enter the cost of assigning task 2 to agent 4: 26

Enter the cost of assigning task 3 to agent 4: 24

Enter the cost of assigning task 4 to agent 4: 10

Optimal Assignment:

Agent 1 is assigned to Task 3

Agent 2 is assigned to Task 1

Agent 3 is assigned to Task 2

Agent 4 is assigned to Task 4

Thus, total Cost is 59.0.

**Algorithm:**

import numpy as np

from scipy . optimize import linear\_sum\_assignment

# Revenue matrix ( convert it into a numpy array )

revenue\_matrix = np . array ([

[32 , 38 , 40 , 28 , 40] , # S1

[40 , 24 , 28 , 21 , 36] , # S2

[41 , 27 , 33 , 30 , 37] , # S3

[22 , 38 , 41 , 36 , 36] , # S4

[29 , 33 , 40 , 35 , 39] # S5

])

# Convert the maximization problem to a minimization

problem

# Subtract all elements from the maximum value in

the

matrix

cost\_matrix = np . max ( revenue\_matrix ) -

revenue\_matrix

# Apply the Hungarian algorithm

row\_indices , col\_indices = linear\_sum\_assignment

( cost\_matrix )

# Optimal assignment and total revenue

optimal\_assignment = list ( zip ( row\_indices ,

col\_indices ) )

total\_revenue = revenue\_matrix [ row\_indices ,

col\_indices ]. sum ()

# Print results

print ( " Optimal Assignment ( Salesperson -> Region ) : " )

for salesperson , region in optimal\_assignment :

print ( f " S { salesperson +1} -> R { region +1} " )

print ( f " Total Maximum Revenue : { total\_revenue } " )

**Output:**

Optimal Assignment (Salesperson → Region):

45

S1 → R2

S2 → R1

S3 → R5

S4 → R3

S5 → R4

Total Maximum Revenue: 191.

**Algorithm:**

def northwest\_corner\_method ( cost\_matrix , supply ,

demand ) :

"""

Solves the Transportation Problem using the

Northwest

Corner Method .

Args :

cost\_matrix ( list of lists ) : Cost matrix where

cost\_matrix [ i ][ j ]

represents the cost of transporting one unit

from

source i to destination j .

supply ( list ) : Supply at each source .

demand ( list ) : Demand at each destination .

Returns :

transportation\_table ( list of lists ) :

Transportation

table representing the optimal transportation

plan .

total\_cost ( float ) : Total cost of

the transportation plan .

"""

# Get the number of sources and destinations

num\_sources = len ( supply )

num\_destinations = len ( demand )

# Initialize the transportation table

transportation\_table = [[0 for \_ in range (

num\_destinations ) ]

for \_ in range ( num\_sources ) ]

print ( " Initial Transportation Table : " )

for row in transportation\_table :

print ( row )

# Initialize the row and column indices

row = 0

col = 0

# Loop until all supply and demand are met

while row < num\_sources and col <

num\_destinations :

print ( f " \ nStep { row +1}: " )

print ( f " Current Row : { row +1} , Current Column

: { col +1} " )

# Calculate the maximum possible allocation

allocation = min ( supply [ row ] , demand [ col ])

print ( f " Allocation : { allocation } units " )

# Update the transportation table

transportation\_table [ row ][ col ] = allocation

print ( " Updated Transportation Table : " )

for r in transportation\_table :

print ( r )

# Update the supply and demand

supply [ row ] -= allocation

demand [ col ] -= allocation

print ( f " Updated Supply at Source { row +1}:

{ supply [ row ]} units " )

print ( f " Updated Demand at Destination { col

+1}:

{ demand [ col ]} units " )

# Move to the next cell

if supply [ row ] == 0:

row += 1

elif demand [ col ] == 0:

col += 1

# Calculate the total cost

total\_cost = sum ( sum ( transportation\_table [ i ][ j ]

\* cost\_matrix [ i ][ j ]

for j in range ( num\_destinations ) )

for i in range ( num\_sources ) )

print ( f " \ nTotal Cost : { total\_cost } " )

return transportation\_table , total\_cost

def main () :

print ( " Transportation Problem Solver " )

# Get the number of sources and destinations

num\_sources = int ( input ( " Enter the number of

sources : " ) )

num\_destinations = int ( input ( " Enter the

number of destinations : " ) )

# Get the cost matrix

cost\_matrix = []

for i in range ( num\_sources ) :

row = []

for j in range ( num\_destinations ) :

cost = float ( input ( f " Enter the cost from source

{ i +1}

to destination { j +1}: " ) )

row . append ( cost )

cost\_matrix . append ( row )

# Get the supply and demand

supply = []

for i in range ( num\_sources ) :

supply\_val = float ( input ( f " Enter the supply at

source { i +1}: " ) )

supply . append ( supply\_val )

demand = []

for i in range ( num\_destinations ) :

demand\_val = float ( input ( f " Enter the demand at

destination { i +1}: " ) )

demand . append ( demand\_val )

# Solve the transportation problem

transportation\_table , total\_cost =

northwest\_corner\_method

( cost\_matrix , supply , demand )

print ( " \ nOptimal Transportation Plan : " )

for row in transportation\_table :

print ( row )

print ( f " Total Cost : { total\_cost } " )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

**Output:**

Transportation Problem Solver:

Enter the number of sources: 3

Enter the number of destinations: 4

Enter the cost from source 1 to destination 1: 5

Enter the cost from source 1 to destination 2: 3

Enter the cost from source 1 to destination 3: 6

Enter the cost from source 1 to destination 4: 2

Enter the cost from source 2 to destination 1: 4

Enter the cost from source 2 to destination 2: 7

Enter the cost from source 2 to destination 3: 9

Enter the cost from source 2 to destination 4: 1

Enter the cost from source 3 to destination 1: 3

Enter the cost from source 3 to destination 2: 4

Enter the cost from source 3 to destination 3: 7

Enter the cost from source 3 to destination 4: 5

Enter the supply at source 1: 19

Enter the supply at source 2: 37

Enter the supply at source 3: 34

Enter the demand at destination 1: 16

Enter the demand at destination 2: 18

Enter the demand at destination 3: 31

Enter the demand at destination 4: 25

Initial Transportation Table:

0 0 0 0

0 0 0 0

0 0 0 0

**Step 1:**

Current Row: 1, Current Column: 1

Allocation: 16.0 unit

Updated Transportation Table:

16.0 0 0 0

0 0 0 0

0 0 0 0

Updated Supply at Source 1: 3.0 units

Updated Demand at Destination 1: 0.0 units

**Step 1:**

Current Row: 1, Current Column: 2

Allocation: 3.0 units

Updated Transportation Table:

16.0 3.0 0 0

0 0 0 0

0 0 0 0

Updated Supply at Source 1: 0.0 units

Updated Demand at Destination 2: 15.0 units

**Step 2:**

Current Row: 2, Current Column: 2

Allocation: 15.0 units

Updated Transportation Table:

16.0 3.0 0 0

0 15.0 0 0

0 0 0 0

Updated Supply at Source 2: 22.0 units

Updated Demand at Destination 2: 0.0 units

**Step 2:**

Current Row: 2, Current Column: 3

Allocation: 22.0 units

Updated Transportation Table:

16.0 3.0 0 0

0 15.0 22.0 0

0 0 0 0

Updated Supply at Source 2: 0.0 units

Updated Demand at Destination 3: 9.0 units

**Step 3:**

Current Row: 3, Current Column: 3

Allocation: 9.0 units

Updated Transportation Table:

16.0 3.0 0 0

0 15.0 22.0 0

0 0 9.0 0

Updated Supply at Source 3: 25.0 units

Updated Demand at Destination 3: 0.0 units

**Step 3:**

Current Row: 3, Current Column: 4

Allocation: 25.0 units

Updated Transportation Table:

16.0 3.0 0 0

0 15.0 22.0 0

0 0 9.0 25.0

Updated Supply at Source 3: 0.0 units

Updated Demand at Destination 4: 0.0 units

Total Cost: 580.0

Optimal Transportation Plan:

16.0 3.0 0 0

0 15.0 22.0 0

0 0 9.0 25.0

Therefore, the total Cost is 580.0.

**Program:**

def least\_cost\_method ( cost\_matrix , supply ,

demand ) :

num\_sources = len ( supply )

num\_destinations = len ( demand )

transportation\_table = [[0 for \_ in range (

num\_destinations ) ]

for \_ in range ( num\_sources ) ]

while sum ( supply ) > 0 and sum ( demand ) > 0:

# Find the minimum cost cell

min\_cost = float ( ’ inf ’)

min\_row = -1

min\_col = -1

for i in range ( num\_sources ) :

for j in range ( num\_destinations ) :

if supply [ i ] > 0 and demand [ j ] >

0 and

cost\_matrix [ i ][ j ] < min\_cost :

min\_cost = cost\_matrix [ i ][ j ]

min\_row = i

min\_col = j

# Allocate as much as possible to the

minimum cost cell

allocation = min ( supply [ min\_row ] , demand

[ min\_col ])

transportation\_table [ min\_row ][ min\_col ] =

allocation

# Update supply and demand

supply [ min\_row ] -= allocation

demand [ min\_col ] -= allocation

# Calculate the total cost

total\_cost = sum ( sum ( transportation\_table [ i

][ j ] \*

cost\_matrix [ i ][ j ] for j in range (

num\_destinations ) )

for i in range ( num\_sources ) )

return transportation\_table , total\_cost

def main () :

print ( " Transportation Problem Solver using

Least Cost Method " )

num\_sources = int ( input ( " Enter the number of

sources : " ) )

num\_destinations = int ( input ( " Enter the

number of destinations : " ) )

cost\_matrix = []

for i in range ( num\_sources ) :

row = []

for j in range ( num\_destinations ) :

cost = float ( input ( f " Enter the cost

from source

{ i +1} to destination { j +1}: " ) )

row . append ( cost )

cost\_matrix . append ( row )

supply = []

for i in range ( num\_sources ) :

supply\_val = float ( input ( f " Enter the

supply at source { i +1}: " ) )

supply . append ( supply\_val )

demand = []

for i in range ( num\_destinations ) :

demand\_val = float ( input ( f " Enter the

demand at

destination { i +1}: " ) )

demand . append ( demand\_val )

# Check if total supply equals total demand

total\_supply = sum ( supply )

total\_demand = sum ( demand )

if total\_supply > total\_demand :

# Add a dummy destination

demand . append ( total\_supply -

total\_demand )

cost\_matrix = [ row + [0] for row in

cost\_matrix ]

# Add a column of zeros for the dummy

destination

num\_destinations += 1

print ( f " Added dummy destination with

demand :

{ total\_supply - total\_demand } " )

elif total\_demand > total\_supply :

# Add a dummy source

supply . append ( total\_demand -

total\_supply )

cost\_matrix . append ([0] \*

num\_destinations )

# Add a row of zeros for the dummy

source

num\_sources += 1

print ( f " Added dummy source with supply :

{ total\_demand - total\_supply } " )

# Solve the transportation problem using the

Least Cost Method

transportation\_table , total\_cost =

least\_cost\_method

( cost\_matrix , supply , demand )

# Print the results

print ( " \ nTransportation Table : " )

for row in transportation\_table :

print ( row )

print ( f " \ nTotal Cost : { total\_cost } " )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

**Output:**

Transportation Problem solver using LCM:

Enter the number of sources: 3

Enter the number of destinations: 4

Enter the cost from source 1 to destination 1: 1

Enter the cost from source 1 to destination 2: 2

Enter the cost from source 1 to destination 3: 3

Enter the cost from source 1 to destination 4: 4

Enter the cost from source 2 to destination 1: 4

Enter the cost from source 2 to destination 2: 3

Enter the cost from source 2 to destination 3: 2

Enter the cost from source 2 to destination 4: 0

Enter the cost from source 3 to destination 1: 0

Enter the cost from source 3 to destination 2: 2

Enter the cost from source 3 to destination 3: 2

Enter the cost from source 3 to destination 4: 1

Enter the supply at source 1: 6

Enter the supply at source 2: 8

Enter the supply at source 3: 10

Enter the demand at destination 1: 4

Enter the demand at destination 2: 6

Enter the demand at destination 3: 8

Enter the demand at destination 4: 6

Transportation table:







0 6.0 0 0

0 0 2.0 6.0

4.0 0 6.0 0







Therefore, Total Cost is 28.0.

**Program:**

def vogels\_approximation\_method ( cost\_matrix , supply ,

demand ) :

num\_sources = len ( supply )

num\_destinations = len ( demand )

transportation\_table = [[0 for \_ in range (

num\_destinations ) ]

for \_ in range ( num\_sources ) ]

while sum ( supply ) > 0 and sum ( demand ) > 0:

row\_penalties = []

col\_penalties = []

for i in range ( num\_sources ) :

if supply [ i ] > 0:

sorted\_costs = sorted ([ cost\_matrix [ i ][ j ]

sfor j in range ( num\_destinations ) if

demand [ j ] > 0])

if len ( sorted\_costs ) > 1:

row\_penalties . append (( sorted\_costs

[1] - sorted\_costs [0] , i ) )

else :

row\_penalties . append (( sorted\_costs

[0] , i ) )

for j in range ( num\_destinations ) :

if demand [ j ] > 0:

sorted\_costs = sorted ([ cost\_matrix [ i ][ j ]

for i in range ( num\_sources ) if supply [ i ]

> 0])

if len ( sorted\_costs ) > 1:

col\_penalties . append (( sorted\_costs

[1] - sorted\_costs [0] , j ) )

else :

col\_penalties . append (( sorted\_costs

[0] , j ) )

max\_row\_penalty = max ( row\_penalties , default =(0 ,

-1) )

max\_col\_penalty = max ( col\_penalties , default =(0 ,

-1) )

if max\_row\_penalty [0] >= max\_col\_penalty [0]:

row = max\_row\_penalty [1]

col = min (( cost\_matrix [ row ][ j ] , j )

for j in range ( num\_destinations ) if demand [ j

] > 0) [1]

else :

col = max\_col\_penalty [1]

row = min (( cost\_matrix [ i ][ col ] , i )

for i in range ( num\_sources ) if supply [ i ] >

0) [1]

allocation = min ( supply [ row ] , demand [ col ])

transportation\_table [ row ][ col ] = allocation

supply [ row ] -= allocation

demand [ col ] -= allocation

total\_cost = sum ( sum ( transportation\_table [ i ][ j ] \*

cost\_matrix [ i ][ j ]

for j in range ( num\_destinations ) ) for i in range (

num\_sources ) )

return transportation\_table , total\_cost

def main () :

print ( " Transportation Problem Solver " )

num\_sources = int ( input ( " Enter the number of sources

: " ) )

num\_destinations = int ( input ( " Enter the number of

destinations : " ) )

cost\_matrix = []

for i in range ( num\_sources ) :

row = []

for j in range ( num\_destinations ) :

cost = float ( input ( f " Enter the cost from

source { i +1}

to destination { j +1}: " ) )

row . append ( cost )

cost\_matrix . append ( row )

supply = []

for i in range ( num\_sources ) :

supply\_val = float ( input ( f " Enter the supply at

source { i +1}: " ) )

supply . append ( supply\_val )

demand = []

for i in range ( num\_destinations ) :

demand\_val = float ( input ( f " Enter the demand at

destination { i +1}: " ) )

demand . append ( demand\_val )

# Check if total supply equals total demand

total\_supply = sum ( supply )

total\_demand = sum ( demand )

if total\_supply > total\_demand :

# Add a dummy destination

demand . append ( total\_supply - total\_demand )

cost\_matrix = [ row + [0] for row in cost\_matrix ]

# Add a column of zeros for the dummy

destination

num\_destinations += 1

print ( f " Added dummy destination with demand :

{ total\_supply - total\_demand } " )

elif total\_demand > total\_supply :

# Add a dummy source

supply . append ( total\_demand - total\_supply )

cost\_matrix . append ([0] \* num\_destinations )

# Add a row of zeros for the dummy source

num\_sources += 1

print ( f " Added dummy source with supply :

{ total\_demand - total\_supply } " )

# Solve the transportation problem

transportation\_table , total\_cost =

vogels\_approximation\_method

( cost\_matrix , supply , demand )

# Print the results

print ( " \ nTransportation Table : " )

for row in transportation\_table :

print ( row )

print ( f " \ nTotal Cost : { total\_cost } " )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

**Output:**

Transportation Problem Solver:

Enter the number of sources: 3

Enter the number of destinations: 4

Enter the cost from source 1 to destination 1: 11

Enter the cost from source 1 to destination 2: 13

Enter the cost from source 1 to destination 3: 17

Enter the cost from source 1 to destination 4: 14

Enter the cost from source 2 to destination 1: 16

Enter the cost from source 2 to destination 2: 18

Enter the cost from source 2 to destination 3: 14

Enter the cost from source 2 to destination 4: 10

Enter the cost from source 3 to destination 1: 21

Enter the cost from source 3 to destination 2: 24

Enter the cost from source 3 to destination 3: 13

Enter the cost from source 3 to destination 4: 10

Enter the supply at source 1: 250

Enter the supply at source 2: 300

Enter the supply at source 3: 400

Enter the demand at destination 1: 200

Enter the demand at destination 2: 225

Enter the demand at destination 3: 275

Enter the demand at destination 4: 250

Transportation Table:







25.0 225.0 0 0

175.0 0 0 125.0

0 0 275.0 125.0







Total Cost: 12075.0.

**Algorithm for Pure Strategy**

import numpy as np

from fractions import Fraction

def get\_payoff\_matrix () :

n = int ( input ( " Enter the number of rows ( n ) : " ) )

m = int ( input ( " Enter the number of columns ( m ) :

" ) )

print ( " Enter the payoff matrix row by row ( space

- separated values ) : " )

payoff\_matrix = []

for i in range ( n ) :

row = input ( f " Row { i + 1}: " ) . split ()

# Convert each input to Fraction

row = [ Fraction ( value ) for value in row ]

if len ( row ) != m :

raise ValueError ( f " Each row must

have exactly { m } values . " )

payoff\_matrix . append ( row )

return np . array ( payoff\_matrix )

def maximin ( payoff\_matrix ) :

# Maximin for Player 1

row\_min = np . min ( payoff\_matrix , axis =1) #

Minimum payoff for each row

maximin\_value = np . max ( row\_min ) # Max of the

row minimums

return maximin\_value

def minimax ( payoff\_matrix ) :

# Minimax for Player 2

col\_max = np . max ( payoff\_matrix , axis =0) #

Maximum payoff for each column

minimax\_value = np . min ( col\_max ) # Min of the

column maximums

return minimax\_value

def main () :

payoff\_matrix = get\_payoff\_matrix ()

maximin\_value = maximin ( payoff\_matrix )

minimax\_value = minimax ( payoff\_matrix )

print ( f " Maximin value for Player 1: {

maximin\_value } " )

print ( f " Minimax value for Player 2: {

minimax\_value } " )

if maximin\_value == minimax\_value :

print ( " The game has a saddle point at

this value . " )

else :

print ( " The game does not have a saddle

point . " )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

**Output:**

Enter the number of rows (n): 2

Enter the number of columns (m): 3

Enter the payoff matrix row by row (space-separated values):

Row 1: 6 8 6

Row 2: 4 12 2

Maximin value for Player 1: 6

Minimax value for Player 2: 6

The game has a saddle point at this value.

**Algorithm for Mixed Strategy**

from fractions import Fraction

def get\_payoff\_matrix () :

print ( " Enter the payoff matrix for a 2 x2 game (4

values ) : " )

matrix = []

for i in range (2) :

row = input ( f " Row { i + 1} ( space -

separated values ) : " ) . split ()

if len ( row ) != 2:

raise ValueError ( " Each row must

have exactly 2 values . " )

matrix . append ([ Fraction ( value ) for value

in row ])

return matrix

def check\_saddle\_point ( matrix ) :

a11 , a12 = matrix [0]

a21 , a22 = matrix [1]

# Check for saddle point

max\_row1 = max ( a11 , a12 )

min\_row2 = min ( a21 , a22 )

max\_col1 = max ( a11 , a21 )

min\_col2 = min ( a12 , a22 )

if max\_row1 == min\_row2 or max\_col1 == min\_col2 :

return True # Saddle point exists

return False # No saddle point

def calculate\_strategies\_and\_value ( matrix ) :

a11 , a12 = matrix [0]

a21 , a22 = matrix [1]

# Calculate the denominator for the mixed

strategy formulas

denominator = ( a11 + a22 ) - ( a12 + a21 )

# Calculate Player A ’s strategy

p1 = ( a22 - a21 ) / denominator

p1 = max (0 , min (1 , p1 ) ) # Ensure p1 is between

0 and 1

p2 = 1 - p1 # Player B ’s strategy

# Calculate Player B ’s strategy

q1 = ( a22 - a12 ) / denominator

q1 = max (0 , min (1 , q1 ) ) # Ensure q1 is between

0 and 1

q2 = 1 - q1 # Player A ’s strategy

# Calculate the value of the game

value\_of\_game = ( a11 \* a22 - a12 \* a21 ) /

denominator

return p1 , p2 , q1 , q2 , value\_of\_game

def main () :

matrix = get\_payoff\_matrix ()

if check\_saddle\_point ( matrix ) :

print ( " Saddle point exists . No need for

mixed strategies . " )

else :

try :

p1 , p2 , q1 , q2 , value\_of\_game =

calculate\_strategies\_and\_value

( matrix )

print ( " Optimal Mixed Strategies :

" )

print ( f " Player A ( Row 1) : { p1 :.4

f } ({ Fraction ( p1 ) .

limit\_denominator () }) " )

print ( f " Player A ( Row 2) : { p2 :.4

f } ({ Fraction ( p2 ) .

limit\_denominator () }) " )

print ( f " Player B ( Column 1) : { q1

:.4 f } ({ Fraction ( q1 ) .

limit\_denominator () }) " )

print ( f " Player B ( Column 2) : { q2

:.4 f } ({ Fraction ( q2 ) .

limit\_denominator () }) " )

print ( f " Value of the Game : {

value\_of\_game :.4 f } ({ Fraction

( value\_of\_game ) .

limit\_denominator () }) " )

except ValueError as e :

print ( e )

if \_\_name\_\_ == " \_\_main\_\_ " :

main ()

**Output:**

Enter the payoff matrix for a 2x2 game (4 values):

Row 1 (space-separated values): 5 1

Row 2 (space-separated values): 3 4

Optimal Mixed Strategies:

Player A (Row 1): 0.2000 (1/5)

Player A (Row 2): 0.8000 (4/5)

Player B (Column 1): 0.6000 (3/5)

Player B (Column 2): 0.4000 (2/5)

Value of the Game: 3.4000 (17/5).