

**Department of Computer Science**

**University *of* Kerala**

Kariavattom Campus

*Accredited by NAAC with A++ Grade*

www.keralauniversity.ac.in

MSc Computer Science

with Specialization in Artificial Intelligence

**CSA-CC-514 Artificial Intelligence Lab**

**Lab Report**

CHRISTY BINU

97422607009

March 2023



**Department of Computer Science**

**University *of* Kerala**

Kariavattom Campus

*Accredited by NAAC with A++ Grade*

www.keralauniversity.ac.in

Certified that this is the bonafide record of the practical work done by .....CHRISTY..BINU........................................ in the **CSA-CC-514 Artificial Intelligence Lab** of First Semester **MSc Computer Science with Specialization in Artificial Intelligence** during the year 2022-2023.

**Faculty - in Charge Head of Department**

**CYCLE-1**

1. Insertion sort

2. Merge sort.

3. String matching.

4. Binary search.

5. N Queen**.**

6. All pair shortest path.

**CYCLE-2**

7. Breadth first search.

8. Depth first search**.**

9. Iterative deepening depth first search.

10. uniform cost search

11. Binary Search Tree

**CYCLE-3**

12. Beam search

13. A star search.

14. Best first search.

15. Alpha beta pruning.

**Maths -Set**

1. Calculate L1 Norm

2. Calculate L2 Norm

3. Calculate Max Norm

4. Hadamard product of two matrix.

5. Define a 3 \* 3 square matrix. Extract the main diagonal as vector. Create the

diagonal matrix from the extracted vector.

6. Find determinant of a matrix.

7. Create an orthogonal matrix and check Q ^ Q = Q Q ^ = I.

8. Find rank of a matrix.

9. Find sparsity of a matrix.

10. Find eigen value and eigen vector Of a Matrix

11. Find eigen values and eigen vectors of a matrix and reconstruct the matrix.

**CYCLE 1**

Algorithm -1

Aim:

To implement Insertion sort

Algorithm:

function insertion\_sort(list: list of sortable items)

size = length(list)

for i = 1 to size - 1 do

currentIndex = i

while currentIndex > 0 and list[currentIndex-1] > list[currentIndex] do

swap(list[currentIndex], list[currentIndex-1])

currentIndex = currentIndex - 1

end while

end for

end function

Output:

How many elements should be inputed?

10

Enter the elements in array :

=>10

=>30

=>2

=>7

=>8

=>34

=>23

=>90

=>10

=>22

[2, 7, 8, 10, 10, 22, 23, 30, 34, 90]

Algorithm -2

Aim:

To implement merge sort

Algorithm:

function mergeSort(array):

if length(array) <= 1:

return array

mid = length(array) // 2

left = mergeSort(array[:mid])

right = mergeSort(array[mid:])

result = []

i = j = 0

while i < length(left) and j < length(right):

if left[i] < right[j]:

result.append(left[i])

i = i + 1

else:

result.append(right[j])

j = j + 1

result += left[i:]

result += right[j:]

return result

Output:

How many elements should be inputed?

5

Enter the elements in array :

=>2

=>4

=>6

=>8

=>22

[2, 4, 5, 6, 8, 22]

Algorithm - 3

Aim:

To implement String pattern matching using naïve method

Algorithm:

NaiveStringMatch(text, pattern):

n = length(text)

m = length(pattern)

i = 0

while i <= n - m do

j = 0

while j < m and pattern[j] = text[i+j] do

j = j + 1

if j = m then

return i

i = i + 1

return -1

Output:

Enter the string: ABcdcagegorysi

Enter the pattern to be searched: cage

Given pattern found in string from position:

4 to 7

Algorithm – 4

Aim:

To implement binary search

Algorithm:

function binarySearch(array, key):

firstElement = 0

lastElement = length(array) - 1

while lastElement >= firstElement:

middleElement = int((firstElement + lastElement) / 2)

if array[middleElement] == key:

return middleElement

elif array[middleElement] > key:

lastElement = middleElement - 1

else:

firstElement = middleElement + 1

return None

Output:

Enter the Number of elements you want to input in an array : 6

Enter the elements in array

>5

>9

>3

>3

>6

>2

Enter the element to be searched : 6

Algorithm – 5

Aim:

To implement n-queen problem

Algorithm:

function attackingAreas(board, row, column):

// Checking column corresponding to that cell

for i from row down to 0:

if board[i][column] == 1:

return False

// Checking upper diagonal corresponding to that cell

for i,j in zip(range(row, -1, -1), range(column, -1, -1)):

if board[i][j] == 1:

return False

// Checking lower diagonal corresponding to that cell

for i, j in zip(range(row, -1, -1), range(column, n)):

if board[i][j] == 1:

return False

return True

function SolveNQueen(board, row):

if row == n:

return True

for i from 0 to n-1:

if attackingAreas(board, row, i) is True:

board[row][i] = 1

if SolveNQueen(board, row + 1) is True:

return True

board[row][i] = 0

return False

Output:

Enter the n - queens to be placed in a n \* n chess board => 4

[[0 1 0 0]

[0 0 0 1]

[1 0 0 0]

[0 0 1 0]]

Algorithm – 6

Aim:

To implement All pair shortest path algorithm

Algorithm:

function all\_pair\_shortest\_path(graph):

v = length(graph)

\\Formula to find the shortest path for all pair of vertices

for k in range(v):

for i in range(v):

for j in range(v):

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

return graph

Output:

Input graph :

0 5 INF 10

INF 0 3 INF

INF INF 0 1

INF INF INF 0

Shortest distance between every pair of vertices :

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**CYCLE 2**

Algorithm – 1

Aim:

To implement Breadth first search

Algorithm:

BFS(graph, startVertex):

queue <- [startVertex]

visited <- {}

traversalList <- []

for each vertex in graph:

visited[vertex] <- False

while queue is not empty:

vertex <- queue.pop(0)

visited[vertex] <- True

traversalList.append(vertex)

for each node adjacent to vertex in graph:

if visited[node] is False and node not in queue:

queue.append(node)

return traversalList

Output:

Input graph : {0: [1, 2, 3], 1: [0, 2], 2: [0, 1, 4], 3: [0], 4: [2]}

Breadth first search :

2 => 0 => 1 => 4 => 3

Algorithm – 2

Aim:

To implement Depth first search

Algorithm:

def DFS(graph, currentNode):

print(currentNode, end=" ")

for node in graph[currentNode]:

if DFS(graph, node):

return True

return False

Output:

Input graph : {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['G'], 'D': [], 'E': ['F'], 'F': [], 'G': []}

Depth first search :

A B D E F C G

Algorithm – 3

Aim:

To implement Iterative Deepening Depth first search

Algorithm:

def IDDFS(graph, currentNode, depth):

for i in range(depth):

if DFS(graph, currentNode, i):

return True

return False

def DFS(graph, currentNode, depth):

print(currentNode, end=" ")

if depth > 0:

for node in graph[currentNode]:

if DFS(graph, node, depth - 1):

return True

else:

return False

Output:

Input graph : {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['G'], 'D': [], 'E': ['F'], 'F': [], 'G': []}

Iterative Deepening Depth first search :

A

A => B => C

A => B => D => E => C => G

A=> B => D => E => F => C => G

Algorithm – 4

Aim:

To implement Iterative Deepening Depth first search

Algorithm:

function uniformCostSearch(problem):

startNode = problem.getStartNode()

frontier = PriorityQueue() // priority queue to store nodes to be expanded

frontier.push((startNode, 0)) // push start node with cost 0

explored = set() // set of visited nodes

while not frontier.isEmpty():

node, pathCost = frontier.pop()

if problem.isGoalNode(node):

return node // return goal node

explored.add(node) // add current node to explored set

for childNode, stepCost in problem.getSuccessors(node):

if childNode not in explored:

frontier.push((childNode, pathCost + stepCost)) // push child node with updated path cost

else:

// update the path cost of the child node in the priority queue

for i, (fNode, fPathCost) in enumerate(frontier.heap):

if fNode == childNode and pathCost + stepCost < fPathCost:

frontier.update(i, (fNode, pathCost + stepCost))

return None // return None if goal node is not found

Output:

input graph and cost : {0: [1, 3], 1: [6], 2: [1], 3: [1, 2, 4], 4: [2, 4], 5: [2, 6], 6: [4]}

cost :

0 => 1 : 2

0 => 3 : 5

1 => 6 : 1

3 => 1 : 5

3 => 6 : 6

3 => 4 : 2

2 => 1 : 4

4 => 2 : 4

4 => 5 : 3

5 => 2 : 6

5 => 6 : 3

6 => 4 : 7

Goal node found....

pathway :

0 => 1 => 6

Minimum Cost of traversal from 0 to 6 = 3

Algorithm – 5

Aim:

To implement a binary search tree and perform insertion, deletion, searching operations

Algorithm:

Insertion on BST :

function insert(root, key, Node):

if root is None:

// If the root is empty, create a new node with the given key

return Node(key)

else if root.value == None:

// If the root's value is None, set it to the given key

root.value = key

else:

if root.value == key:

// If the root's value is the same as the given key, return the root

return root

else if root.value > key:

// If the root's value is greater than the given key, insert it into the left subtree

root.left = insert(root.left, key, Node)

else:

// If the root's value is less than the given key, insert it into the right subtree

root.right = insert(root.right, key, Node)

// Return the root after the insertion

return root

Deletion on BST:

function delete(root, key):

if root is None:

// Key not found in the tree, return None

return None

if key < root.value:

// Key is in the left subtree

root.left = delete(root.left, key)

else if key > root.value:

// Key is in the right subtree

root.right = delete(root.right, key)

else:

// Key is found in the root

if root.left is None:

// Replace root with right child

temp = root.right

root = temp

else if root.right is None:

// Replace root with left child

temp = root.left

root = temp

else:

// Find the minimum value node in the right subtree

temp = root.right

while temp.left is not None:

temp = temp.left

// Replace root value with minimum value node value

root.value = temp.value

// Delete the minimum value node

root.right = delete(root.right, temp.value)

return root

Searching a node in BST

function search(root, key):

if root is None:

// Key not found in the tree

return False

if root.value == key:

// Key found in the root node

return True

else if root.value > key:

// Key may be present in the left subtree

return search(root.left, key)

else:

// Key may be present in the right subtree

return search(root.right, key)

Output:

Insertion

Enter the node to be inserted => 100

Enter the node to be inserted => 20

Enter the node to be inserted => 13

Enter the node to be inserted => 50

Enter the node to be inserted => 300

Enter the node to be inserted => 113

Inorder traversal of created binary tree

13 20 50 100 113 300

Deletion

Enter the element to be deleted : 50

Specified element has been deleted

Inorder traversal

13 20 100 113 300

Search a node

Enter the element to be searched : 113

Element found in the given tree

Enter the element to be searched : 12

Element not found in the given tree

**CYCLE 3**

Algorithm – 1

Aim:

To perform beam search on graph

Algorithm:

function beam\_search(start\_state, beam\_width, max\_depth):

# Initialize the beam with the start state

beam = [start\_state]

# Keep track of the best candidate solution found so far

best\_solution = None

# Loop until we have explored up to the maximum depth or run out of candidates

for depth in range(max\_depth):

# Create an empty list to hold the new candidates generated

candidates = []

# Loop over all the states in the current beam

for state in beam:

# Generate all the possible next states from this state

next\_states = generate\_next\_states(state)

# Add each of these states to the list of candidates

candidates.extend(next\_states)

# Sort the candidates by their scores (in descending order)

candidates.sort(key=score, reverse=True)

# Truncate the list of candidates to the beam width

candidates = candidates[:beam\_width]

# Check if any of the candidates is a solution

for candidate in candidates:

if is\_solution(candidate):

# Update the best solution found so far

best\_solution = candidate

# Replace the current beam with the list of candidates

beam = candidates

# Return the best solution found (if any)

return best\_solution

Output:

Beam Search

input graph : {0: [1, 3], 1: [6], 2: [1], 3: [1, 4, 6], 4: [2, 4], 5: [2, 6], 6: [4]}

Starting node : 0

Goal node : 6

cost :

0 => 1 : 2

0 => 3 : 5

1 => 6 : 1

3 => 1 : 5

3 => 6 : 6

3 => 4 : 2

2 => 1 : 4

4 => 2 : 4

4 => 5 : 3

5 => 2 : 6

5 => 6 : 3

6 => 4 : 7

Enter the heuristics from 0 to Goal : 4

Enter the heuristics from 1 to Goal : 3

Enter the heuristics from 2 to Goal : 5

Enter the heuristics from 3 to Goal : 2

Enter the heuristics from 4 to Goal : 5

Enter the heuristics from 5 to Goal : 6

Enter the beam width : 2

Goal node found....

pathway : 0 => 3 => 6

Cost of Traversal :

11

Algorithm – 2

Aim:

To implement A\* search algorithm

Algorithm:

function A\_Star(start, goal):

// Initialization

openSet := {start}

closedSet := {}

gScore := map with default value of infinity

gScore[start] := 0

fScore := map with default value of infinity

fScore[start] := heuristic\_cost\_estimate(start, goal)

cameFrom := {}

// Main loop

while openSet is not empty:

// Find node with the lowest fScore

current := node in openSet with lowest fScore

if current = goal:

return reconstruct\_path(cameFrom, current)

// Move node to closed set

openSet.remove(current)

closedSet.add(current)

// Check neighbors

for neighbor in get\_neighbors(current):

if neighbor in closedSet:

continue // Ignore already evaluated nodes

tentative\_gScore := gScore[current] + distance\_between(current, neighbor)

if neighbor not in openSet:

openSet.add(neighbor)

else if tentative\_gScore >= gScore[neighbor]:

continue // This path is worse than the previous path

// This path is the best until now, record it!

cameFrom[neighbor] := current

gScore[neighbor] := tentative\_gScore

fScore[neighbor] := gScore[neighbor] + heuristic\_cost\_estimate(neighbor, goal)

// Open set is empty but goal was never reached

return failure

Output:

A\* search Algorithm

Input graph and cost : {0: [1, 3], 1: [6], 2: [1], 3: [1, 4, 6], 4: [2, 4], 5: [2, 6], 6: [4]}

cost :

0 => 1 : 2

0 => 3 : 5

1 => 6 : 1

3 => 1 : 5

3 => 6 : 6

3 => 4 : 2

2 => 1 : 4

4 => 2 : 4

4 => 5 : 3

5 => 2 : 6

5 => 6 : 3

6 => 4 : 7

Enter the heuristics from 0 to Goal : 4

Enter the heuristics from 1 to Goal : 3

Enter the heuristics from 2 to Goal : 5

Enter the heuristics from 3 to Goal : 2

Enter the heuristics from 4 to Goal : 5

Enter the heuristics from 5 to Goal : 6

Goal node found....

pathway :

0 => 1 => 6

Cost of Traversal :

3

Algorithm – 3

Aim:

To implement Beam search algorithm

Algorithm:

function Beam\_Search(start, beam\_width):

// Initialization

openSet := {start}

cameFrom := {}

gScore := map with default value of infinity

gScore[start] := 0

// Main loop

while openSet is not empty:

// Expand the nodes in the current level

currentLevel := {}

for current in openSet:

currentLevel.add(current)

openSet.clear()

// Check each node in the current level

for current in currentLevel:

if is\_goal(current):

return reconstruct\_path(cameFrom, current)

// Check each neighbor

for neighbor in get\_neighbors(current):

tentative\_gScore := gScore[current] + distance\_between(current, neighbor)

if neighbor not in gScore or tentative\_gScore < gScore[neighbor]:

// This is the best path to this node so far

cameFrom[neighbor] := current

gScore[neighbor] := tentative\_gScore

// Add the neighbor to the next level

openSet.add(neighbor)

// If there are more than beam\_width nodes in the next level,

// keep only the top beam\_width nodes according to their gScore

if size(openSet) > beam\_width:

openSet := select\_top\_beam\_width\_nodes(openSet, gScore)

// Open set is empty but goal was never reached

return failure

Output:

Best first Search

input graph : {0: [1, 3], 1: [6], 2: [1], 3: [1, 4, 6], 4: [2, 4], 5: [2, 6], 6: [4]}

Starting node : 0

Goal node : 6

cost :

0 => 1 : 2

0 => 3 : 5

1 => 6 : 1

3 => 1 : 5

3 => 6 : 6

3 => 4 : 2

2 => 1 : 4

4 => 2 : 4

4 => 5 : 3

5 => 2 : 6

5 => 6 : 3

6 => 4 : 7

Enter the heuristics from 0 to Goal : 4

Enter the heuristics from 1 to Goal : 3

Enter the heuristics from 2 to Goal : 5

Enter the heuristics from 3 to Goal : 2

Enter the heuristics from 4 to Goal : 5

Enter the heuristics from 5 to Goal : 6

Goal node found....

pathway :

0 => 3 => 6

Cost of Traversal :

11

Algorithm – 4

Aim:

To implement Alpha beta pruning

Algorithm:

function minimax(node, depth, is\_maximizing\_player, alpha, beta):

if node is a leaf node:

return value of the node

if is\_maximizing\_player:

best\_value = -INFINITY

for each child node:

value = minimax(child node, depth+1, false, alpha, beta)

best\_value = max(best\_value, value)

alpha = max(alpha, best\_value)

if beta <= alpha:

break

return best\_value

else:

best\_value = +INFINITY

for each child node:

value = minimax(child node, depth+1, true, alpha, beta)

best\_value = min(best\_value, value)

beta = min(beta, best\_value)

if beta <= alpha:

break

return best\_value

Output:

The terminal nodes :

3 5 6 9 1 2 0 -1

The optimal value is : 5

**Mathematics Questions**

Algorithm – 1

Aim:

Calculate L1 Norm of a vector.

Algorithm:

function l1\_norm(vector):

initializing variables

loop until index < len(vector):

element = vector[index]

norm = norm + abs(element)

index = index + 1

return norm

Output:

Enter the values of vector :

1

2

3

The L1 norm of vector is : 6

Algorithm – 2

Aim:

Calculate L2 Norm of a vector.

Algorithm:

function l2\_norm(vector):

initializing variables

loop until index < len(vector):

element = vector[index]

norm = norm + square(element)

index = index + 1

return squareroot(norm)

Output:

Enter the values of vector :

1

2

3

The L2 norm of vector is : 3.7416573867739413

Algorithm – 3

Aim:

Calculate Max Norm of a vector.

Algorithm:

function max\_norm(vector):

initializing variables

loop until index < len(vector):

element = vector[index]

if absoluteValue(element) > norm :

norm = absoluteValue(element)

return norm

Output:

Enter the values of vector :

1

2

3

The Max norm of vector is : 3

Algorithm – 4

Aim:

Hadamard product of two matrix.

Algorithm:

function hadamard\_product(matrix1, matrix2):

initializing result array

loop i from 0 to length(matrix1):

row = []

loop j from 0 to length(matrix2):

row.add(matrix1[i][j] \* matrix2[i][j])

result.add(row)

return result array

Output:

Enter the number of rows in Matrices : 2

Enter the number of columns in Matrices : 2

Enter the elements of the first 2x2 matrix

=>1

=>2

=>3

=>4

Enter the elements of the second 2x2 matrix

=>5

=>6

=>7

=>8

first Matrix

[[1 2]

[3 4]]

Second Matrix

[[5 6]

[7 8]]

Hadamard product of two matrices are:

[[ 5 12]

[21 32]]

Algorithm – 5

Aim:

Define a 3 \* 3 square matrix. Extract the main diagonal as vector. Create the

diagonal matrix from the extracted vector.

Algorithm:

function vectorExtractor(matrix):

initializing vector array

loop i from 0 to length of row of matrix:

loop j from 0 to length of column of matrix:

if i == j:

add matrix[i][j] to vector array

return vector array

function diagonalMatrix():

matrix = vectorExtractor(matrix)

initializing diagonal matrix

loop i from 0 to length of row of matrix:

loop j from 0 to length of column of matrix:

if i == j:

add matrix[i][j] to diagonal matrix

else :

add 0 to diagonal matrix

Output:

Enter the elements of the 3x3 matrix

=>12

=>3

=>5

=>6

=>7

=>8

=>9

=>4

=>2

Matrix

[[12 3 5]

[ 6 7 8]

[ 9 4 2]]

Vector

[12 7 2]

Diagonal Matrix

[12 0 0]

[ 0 7 0]

[ 0 0 2]

Algorithm – 6

Aim:

Find determinant of a matrix.

Algorithm:

function determinantOfMatrix(matrix, dimension):

if(dimension < 3):

find determinant using determinant2D function

else:

determinant = 0

loop k from 0 to length of matrix[0]:

array = []

loop i from 0 to dimension:

loop j from 0 to dimension:

if i == 0 or j == k: //for selecting minor of first three elements of matrix

continue

else :

add matrix[i][j] to minor array

if k % 2 == 0 :

determinant += matrix[0][k] \* determinantOfMatrix(minor array, length of minor array)

else :

determinant -= matrix[0][k] \* determinantOfMatrix(minor array, length of minor array)

return determinant

// function to find determinant of matrix less than 3 dimension

function determinant2D(matrix):

initializing variables determinant, diagonal1 and diagonal2

for i in from 0 to len(matrix):

for j in from 0 to len(matrix):

// multiplying two diagonals

if i == j :

diagonal1 \*= matrix[i][j]

else :

diagonal2 \*= matrix[i][j]

determinant = diagonal1 - diagonal2

return determinant

Output:

Elements of matrix:

[[1 2 3]

[4 5 6]

[7 8 9]]

determinant of matrix:

0

Algorithm – 7

Aim:

Create an orthogonal matrix and check Q ^ Q = Q Q ^ = I.

Algorithm:

function VerifyOrthogonal(orthogonal matrix):

let Q be orthogonal matrix

taking transpose of Q and naming it Qtranspose

Q\*Qtranspose = product(Q, Qtranspose)

Qtranspose \* Q = product(Qtranspose, Q)

if Q\*Qtranspose == Qtranspose\*Q == identityMatrix :

return True

else :

return False

Output:

Matrix :

[ 0.33333333 0.66666667 -0.66666667]

[-0.66666667 0.66666667 0.33333333]

[ 0.66666667 0.33333333 0.66666667]

Transpose of matrix :

[ 0.33333333 -0.66666667 0.66666667]

[ 0.66666667 0.66666667 0.33333333]

[-0.66666667 0.33333333 0.66666667]

Q \* Qtranspose :

[1. 0. 0.]

[0. 1. 0.]

[0. 0. 1.]

Qtranspose \* Q :

[1. 0. 0.]

[0. 1. 0.]

[0. 0. 1.]

Here we can see that Q \* Qtranspose = Qtranspose \* Q =

[1 0 0]

[0 1 0]

[0 0 1]

Algorithm – 8

Aim:

Find rank of a matrix.

Algorithm:

function rank(matrix):

n = number of rows in matrix

m = number of columns in matrix

rank = 0

for j from 1 to m do

pivot\_row = -1

for i from 1 to n do

if matrix[i, j] != 0 then

pivot\_row = i

break

if pivot\_row != -1 then

rank = rank + 1

if pivot\_row != 1 then

swap row 1 with pivot\_row

for i from 2 to n do

if matrix[i, j] != 0 then

row\_factor = matrix[i, j] / matrix[1, j]

for k from 1 to m do

matrix[i, k] = matrix[i, k] - row\_factor \* matrix[1, k]

return rank

Output :

Matrix :

[1 2 3]

[4 5 6]

[7 8 9]

Rank of matrix is : 3

Algorithm – 9

Aim:

Find sparsity of a matrix.

Algorithm

function sparcity0fMatrix(matrix):

initializing of variables

the variable count used to count the number of zeros in matrix

for row in matrix:

for element in row :

if element == 0:

count = count + 1

rows = length of matrix

columns = length of first index of matrix

TotalElements = rows \* columns

sparcity = count / TotalElements

return sparcity

Output:

[[5 0 3]

[0 0 6]

[2 0 0]]

Sparcity of given matrix is : 0.5555555555555556

The given Matrix is a sparse matrix

Algorithm – 10

Aim:

Find eigen value and eigen vector Of a Matrix

Algorithm

1. Given a square matrix A of size nxn, calculate the characteristic polynomial p(x) = det(A - x\*I) where I is the identity matrix of size nxn.

2. Solve the characteristic polynomial p(x) for its roots, which are the eigenvalues of A.

3. For each eigenvalue lambda, calculate the null space of the matrix (A - lambda\*I). These null spaces are the eigenvectors of A corresponding to the eigenvalue lambda.

4. Normalize each eigenvector found in step 3 to have unit length.

5. The eigenvalues and eigenvectors of A are the values and vectors found in steps 2 and 3, respectively.

Output:

Input matrix :

[[ 1 2 3]

[ 4 2 1]

[ 3 5 12]]

Eigen value of a matrix :

[13.67489478 -0.93733814 2.26244336]

Eigen vector of a matrix :

[-0.25207554 -0.55072961 -0.17900252]

[-0.16799434 0.81353407 -0.851715 ]

[-0.95301407 -0.18670622 0.49248315]

Algorithm – 11

Aim:

Print Eigen Values and eigen vectors of a matrix and reconstruct the matrix

Algorithm

1. find eigen values and eigen vectors of matrix

2. Take the inverse of eigen vector

3. Use the eigen values to create a diagonal matrix

4. reconstruct matrix by taking dot product of eigen vector, Diagonal matrix created using eigen values and inverse of eigen vector

Output:

Input matrix :

[ 1 2 3]

[ 4 2 1]

[ 3 5 12]

eigen value of matrix is :

[13.67489478 -0.93733814 2.26244336]

eigen vector of matrix is :

[-0.25207554 -0.55072961 -0.17900252]

[-0.16799434 0.81353407 -0.851715 ]

[-0.95301407 -0.18670622 0.49248315]

Reconstructed matrix is :

[ 1. 2. 3.]

[ 4. 2. 1.]

[ 3. 5. 12.]