Output:

cat (x) → x = fubby

spot (fubby, x) → x = black

spot (figaro, x) → x = white

owns (mary, fubby) → True

loves (mary, y) → y = fubby

DATE

EXPT. NO. 12

PAGE NO.

Facts: Fubby is a cat. Fubby has black spots. Figaro is a dog. Figaro has white spots.

Rules: Mary owns a pet if it is a cat and it has black spots. If someone owns something, he loves it.

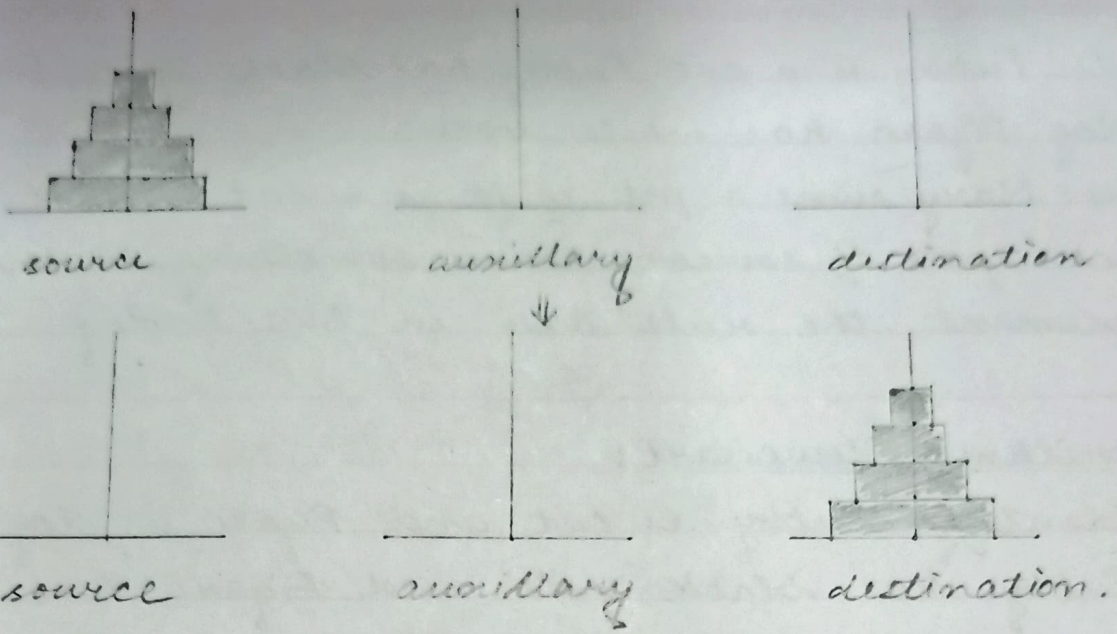
Implement the rule base in SWI-Prolog.

Algorithm / Flowchart:

i) Identify Fubby as cat and Figaro as dog.
ii) Fubby has black spots and Figaro has white.
iii) Identify Mary owns a cat or a dog
iv) Check if its a cat and it has black spots.
v) If Mary owns that, she loves it.

Programme Listing:

```
cat(fubby).
dog(figaro).
spot(fubby, black).
spot(figaro, white).
owns(mary, Y): cat(Y), spot(Y, Z), Z = black.
loves(X, Y) : owns(X, Y).
```



Towers of Hanoi

Output:

$n = 3$

Move disk 1 from tower A to tower C
Move disk 2 from tower A to tower B
Move disk 1 from tower C to tower B
Move disk 3 from tower A to tower C
Move disk 1 from tower B to tower A
Move disk 2 from tower B to tower C
Move disk 1 from tower A to tower C

DATE

EXPT. NO. 13

pioneerpaper.co
PAGE NO.

Implement the Tower of Hanoi using SWI Prolog & Python.

Algorithm / Flowchart:

i) Move n-1 disk from source to auxillary
ii) Move n^{th} disk from source to destination
iii) Move n-1 disk from auxillary to destination

Procedure toh (disk, source, dest, aux)
if (disk == 1), move source's disk to destination, or,
toh (disk-1, source, aux, dest)
move disk (source to dest)
toh (disk-1, aux, dest, source)

Programme Listing:

Python:

```
def toh (n, source, dest, aux)
    if (n==1): print("Move disk 1 from tower", source,
                     "to tower", dest)
    return
    toh (n-1, source, aux, dest)
    print ("Move disk", n, "from rod", source, "to rod", dest)
    toh (n-1, aux, dest, source)
n = 5/1/2/3/...
toh (n, 'A', 'B', 'C')
```

Output:

```
!? → move (4, source, target, aux)
move disk from source to aux
move disk from source to target
move disk from aux to target
move disk from source to aux
move disk from target to source
move disk from target to aux
move disk from source to aux
move disk from source to target
move disk from aux to target
move disk from aux to source
move disk from target to source
move disk from aux to target
move disk from source to aux
move disk from source to target
move disk from aux to target
```

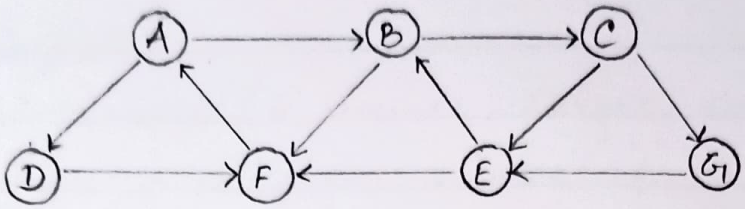
DATE

EXPT. NO.

pioneerpaper.co
PAGE NO.

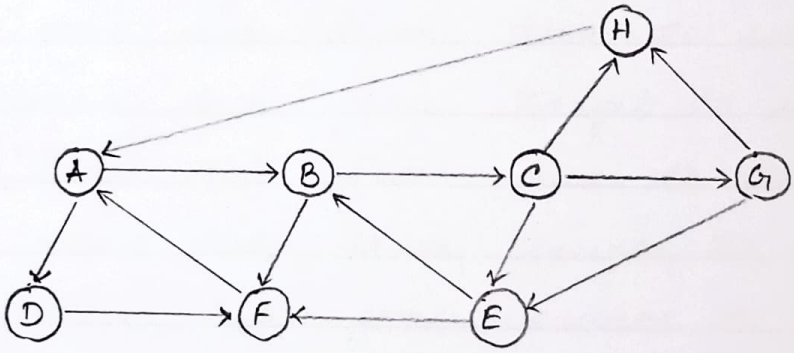
Swi Prolog:

```
move(1,X,Y,_):-
    write('Move disk from'),write(X),write('to'),nl.
        write(Y)
move(N,X,Y,Z)
    N>1
    M is N-1
    move(M,X,Z,Y)
    move(1,X,Y,_)
    move(M,Z,X,Y)
```



BFS (Breadth First Search)

Adjacency Lists => A : B, D
B : C, F
C : E, G1
E : B, F
G1 : E
F : A
D : F



DFS (Depth First Search)

Adjacency Lists => A : B, D
B : C, F
C : E, G1, H
E : B, F
G1 : E, H
F : A
D : F
H : A

DATE

EXPT. NO. 14

pioneerpaper.co
PAGE NO.

Implement BFS and DFS using Python.

Algorithm / Flowchart:

BFS

i) Set status of every node in G as -1.

ii) Enqueue the starting node A and start after setting its status = 2 (waiting state)

iii) Repeat steps (iv) and (v) until queue is empty.

iv) Dequeue a node N, process it and set status = 3

v) Enqueue all neighbours of N that are in ready state (status = 1) and set their status = 2.

vi) Exit

DFS

i) Set status of each node in G as 1.

ii) Push starting node A of the stack and set its status = 2 (waiting state)

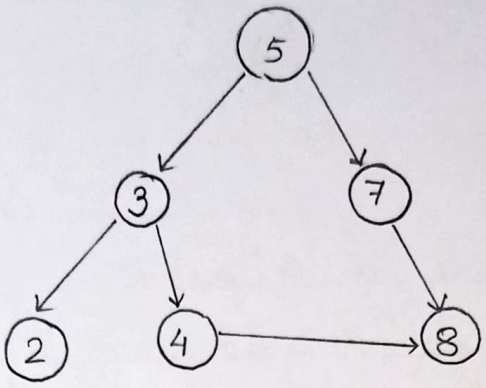
iii) Repeat steps (iv) and (v) until stack is empty.

iv) Pop the top node N, process it and set its status = 3 (proced stp)

v) Push on the stack all the neighbours of N that are in the ready state (status = 1) and set their status = 2.

vi) Exit

PIONEER®



Output:

Following is BFS

5 3 7 2 4 8

DATE

EXPT. NO.

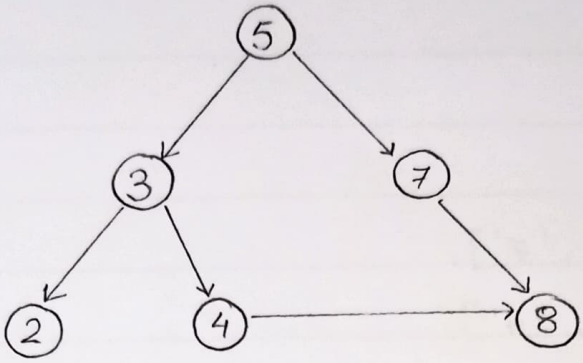
PAGE NO.

Programme Listing:

BFS

```
graph = {
        '5' : ['3','7'],
        '3' : ['2', '4'],
        '7' : ['8'],
        '2' : [ ],
        '4' : ['8'],
        '8' : [ ] }
visited = [ ]
queue = [ ]
def bfs (visited, graph, node):
        visited.append (node)
        queue. append (node)
        while queue:
            m = queue.pop (0)
            print (m, end = " ")
            for neighbour in graph [m]:
                if neighbour not in visited:
                    visited.append (neighbour)
                    queue. append (neighbour)
print ("Following is BFS")
bfs (visited, graph, '5')
```

PIONEER©



Output:

Following is DFS

5 3 2 4 8 7

DFS

```
graph = {
        '5' : ['3', '7'],
        '3' : ['2', '4'],
        '7' , ['8'],
        '2' , [ ],
        '4' , ['8'],
        '8', [] }
visited = set()
def dfs (visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs (visited, graph, neighbour)
print ("Following is DFS")
dfs (visited, graph, '5')
```

DATE

EXPT. NO. 15

pioneerpaper.co
PAGE NO.

Implementation of A* Algorithm.

Algorithm / Flowchart:

i) Place the starting node in OPEN list.

ii) Check if the open list is empty or not, if empty then return failure and stop.

iii) Select the node from the "open" list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop.

iv) Otherwise expand node N and generate all its successors and put n into closed list for each successor n, check whether n is already in 'open' or 'closed', then if absent, compute evaluation function for n and place it into 'open' list.

v) Else if node n is already in 'open' list and 'closed' list, then it should be attached to back pointer which reflects the lowest g(n') value.

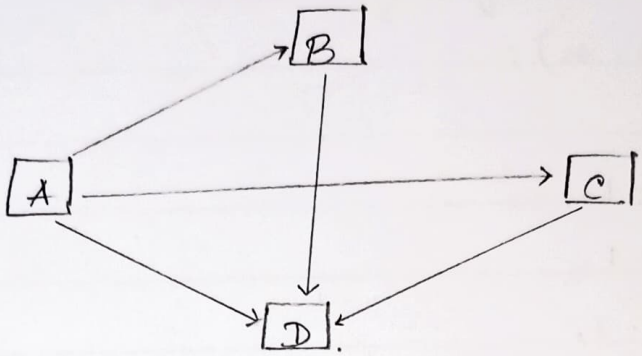
vi) Repeats steps from (ii).

Programme Listing:

```
from collections import deque
class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis
    def get_neighbours(self, v):
```

PIONEER®

```
        return self.adjac_lis[v]
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]
    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
            n = None
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n]
                                        + self.h(n):
                    n = v;
            if n == None:
                print('Path does not exist!')
                return None
```



Output:

```
adjac_lis = {
        'A' :[('B', 1), ('C', 3), ('D', 7)],
        'B' : [('D', 5)],
        'C' : [('D', 12)] }
        graph1 = graph(adjac_lis)
        graph1.a_star_algorithm('A', 'D')
```

path found : ['A', 'B', 'D']

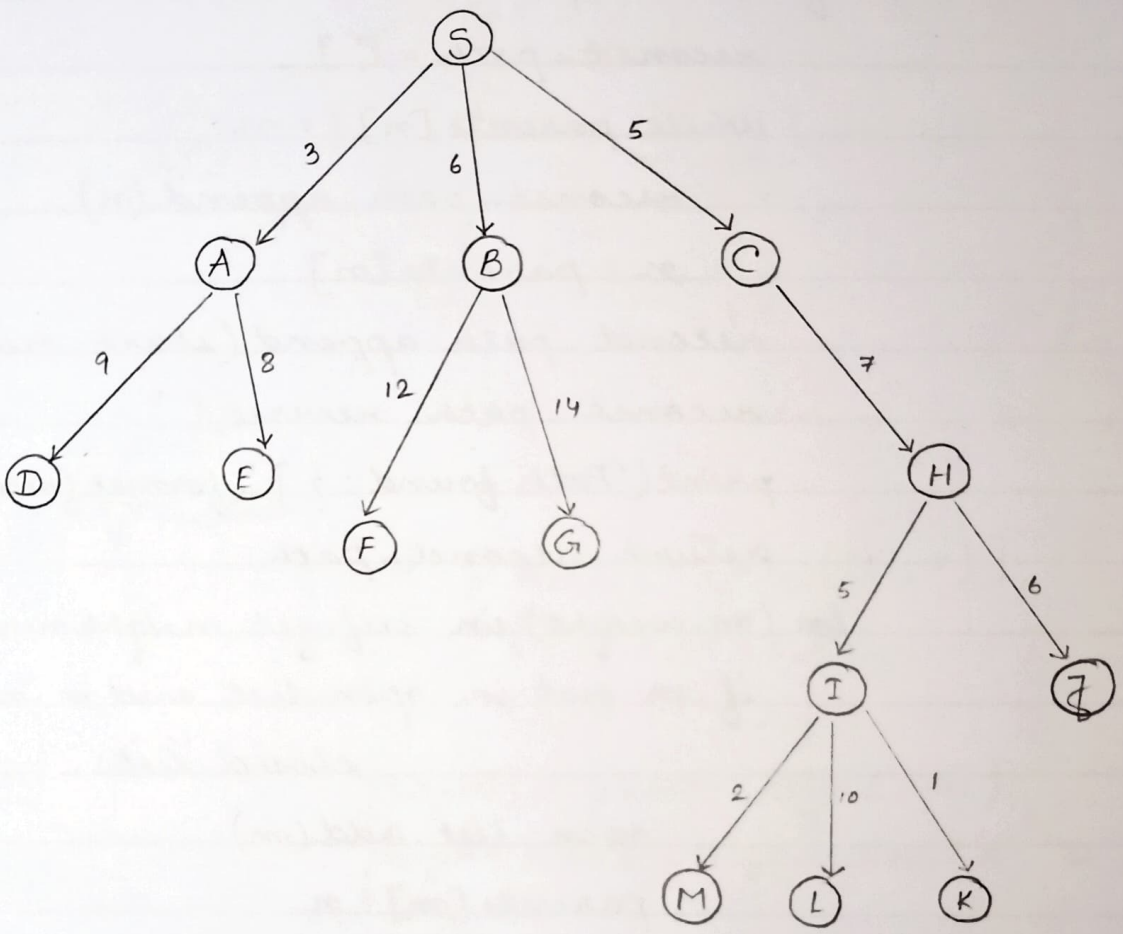
DATE

EXPT. NO.

pioneerpaper.co
PAGE NO.

```
if n == stop_node
    reconst_path = [ ]
    while parents[n] != n:
        reconst_path.append(n)
        n = parents[n]
    reconst_path.append(start_node)
    reconst_path.reverse( )
    print('Path found: {}'.format(reconst_path))
    return reconst_path
for (m, weight) in self.get_neighbours(n):
    if m not in open_list and m not in
                                closed list:
        open_list.add(m)
        parents[m] = n
        g[m] = g[n] + weight
    else:
        if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n
            if m in closed_list:
                closed_list.remove(m)
                open_list.add(m)
    open_list.remove(n)
    closed_list.~~remove~~ add(n)
print('Path does not exist!')
return none
```

PIONEER©


S
3
6
5
A
B
C
9
8
12
14
7
D
E
F
G
H
5
6
I
J
2
10
1
M
L
K

DATE

EXPT. NO. 16

PAGE NO.

Implementation of Greedy Algorithm using Python.
(Best First Search [Informed Search])

Algorithm / Flowchart:

i) Create two empty lists : OPEN and CLOSED

ii) Start from initial node n and put it in the ordered OPEN list.

iii) Repeat the next steps until GOAL node is reached.

iv) If the open list is empty, then exit the loop and return false.

v) Select the top node n in the open list and move it to closed list, also store info of parent node.

vi) If n is a GOAL node, then move the node to the closed list and exit the loop after returning true. The solution is formed by backtracking.

vii) If n is not the GOAL node, expand node n to generate the 'immediate' next node linked to n and add them to open list.

viii) Reorder the nodes to open list in ascending order according to an evolution function.

DATE

EXPT. NO.

pioneerpaperco.com
PAGE NO.

Programme Listing:

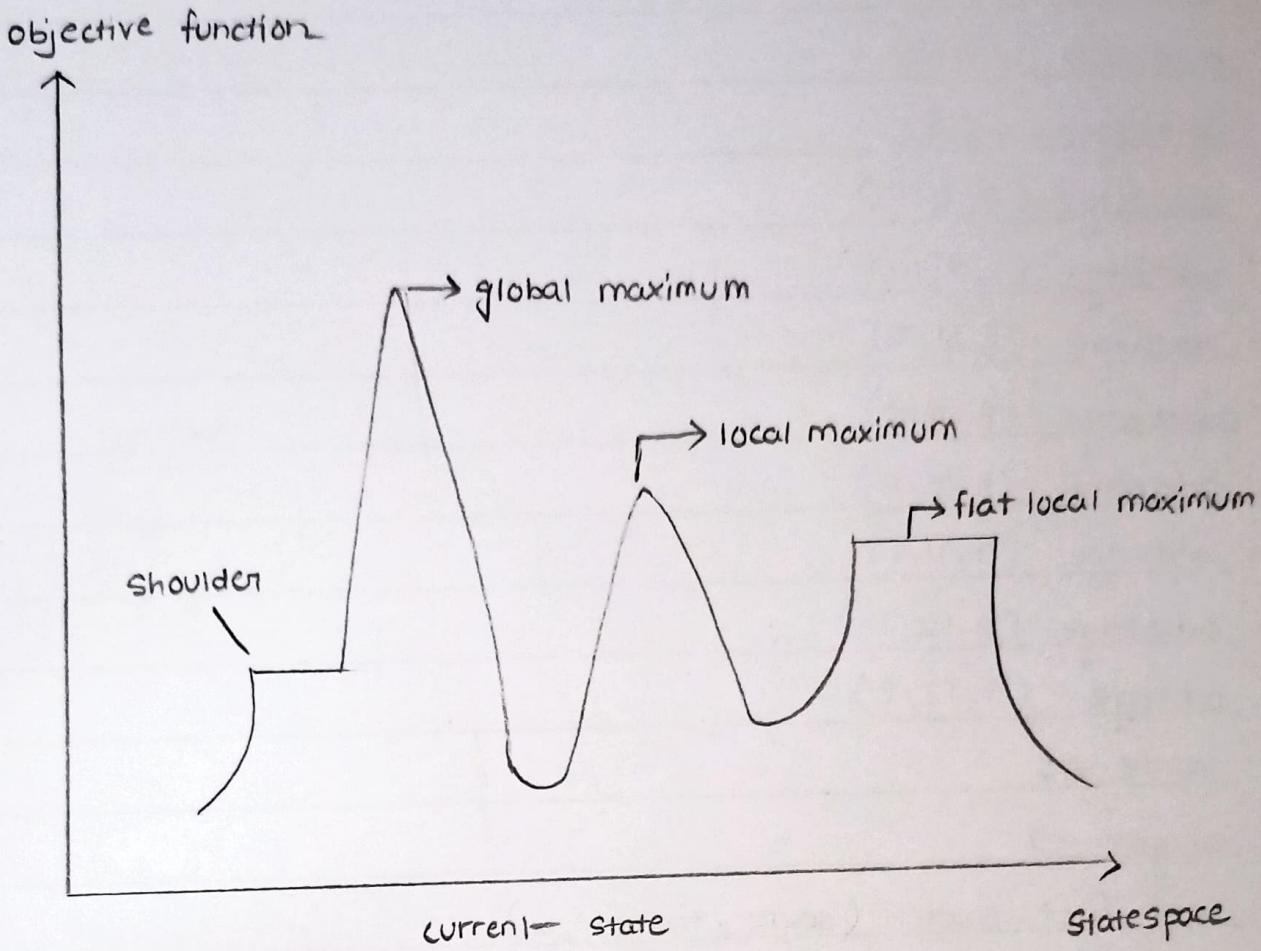
```
from queue import PriorityQueue
v = 14
graph = [[] for i in range = (v)]
def best_first_search (actual_src, target, n):
        visited = [False] * n
        pq = PriorityQueue()
        pq.put ((0, actual_src))
        visited [actual_src] = true
        while pq.empty() == False:
            u = pt.get()[1]
            print(u, end = " ")
            if u == target:
                break
            for v, c in graph [4]:
                if visited [v] == False:
                        visited [v] = True
                        pq.put((c, v))
            print()
def addedge (x, y, cost):
            graph [x].append ((y, cost))
            graph [y].append ((x, cost))
addedge (0, 1, 3)
addedge (0, 2, 6)
addedge (0, 3, 5)
```

PIONEER©

Output:

0 1 3 2 8 9

addedge (1, 4, 9)
addedge (1, 5, 8)
addedge (2, 6, 12)
addedge (2, 7, 14)
addedge (3, 8, 7)
addedge (8, 5, 9)
addedge (8, 10, 6)
addedge (9, 11, 1)
addedge (9, 12, 10)
addedge (9, 13, 2)
source = 0
target = 9
best_first_search (source, target, v)


objective function
global maximum
local maximum
flat local maximum
shoulder
current state
statespace

DATE

EXPT. NO. 17

PAGE NO.

Hill Climbing Algorithm using Python –

Algorithm / Flowchart :-

i) Evaluate the initial state, if it is goal state then return success and stop.

ii) Loop until a solution is found or there is no new operator left to apply.

iii) Select and apply an operator to current state.

iv) Check new state, ie, if goal state, return success and quit, else if better than current state, then assign it as current state or not any better, return to step (ii)

Program listing :-

```
import random
import numpy as np
import networkx as nx
coordinate = np.array([[1,2],[30,21],[56,23],[8,18],[20,50],
                       [3,4],[1.6],[6,7],[15,20],[10,9],[12,12]])
def generate_matrix(coordinate):
    for matrix = []
        for i in range(len(coordinate)):
            for j in range(len(coordinate)):
                p = np.linalg.norm(coordinate[i] - coordinate[j])
                matrix.append(p)
        matrix = np.reshape(matrix(len(coordinate), len(coordinate)))
```

PIONEER®

```
        return matrix
def solution (matrix) :
    points = list(range (0, len(matrix)))
    solution = []
    for i in range (0, len(matrix)) :
        random_point = points [random.randint (0, len(points) - 1)]
        solution.append (random_point)
        points.remove (random_point)
    return solution
def path_length (matrix, solution) :
    cycle_length = 0
    for i in range (0, len(solution)) :
        cycle_length += matrix [solution[i] Solution [i-1]]
        return cycle_length
def neighbours (matrix, solution) :
    neighbours = []
    for i in range (len(solution)) :
        for j in range (i+1, len(solution)) :
            neighbours = solution.copy()
            neighbours [i] = solution [j]
            neighbours [j] = solution [i]
            neighbours.append (neighbours)
    best_neighbour = neighbour [0]
    best_path = path_length (matrix, best_neighbour)
    for neighbour in neighbours :
```

Output :

The solution is :

[2, 4, 8, 3, 7, 5, 0, 6, 9, 10, 1]

```
        current_path = path_length(matrix, neighbour)
        if current_path < best_path
            best_path = current_path
            best_neighbour = neighbour
    return best_neighbour, best_path
def hill_climbing(coordinate):
    matrix = generate_matrix(coordinate)
    current_solution = solution(matrix)
    current_path = path_length(matrix, current_solution)
    neighbour = neighbours(matrix, current_solution)[0]
    best_neighbour, best_neighbour_path = neighbours
                                          (matrix, neighbour)
    while best_neighbour_path < current_path :
        current_solution = best_neighbour
        current_path = best_neighbour_path
        neighbour = neighbours(matrix, current_solution)
        best_neighbour, best_neighbour_path =
            neighbours(matrix, neighbour)
    return current_path, current_solution
final_solution = hill_climbing(coordinate)
print("The solution is \n", final_solution[1])
```

DATE

EXPT. NO. 18

PAGE NO.

4 Queens Problem using Python :

Algorithm / Flowchart :-

i) Start in the left most column.

ii) If all queens are placed, return true.

iii) Try all the rows in the current column.

iv) a) If the queen can be placed safely in this row then mark this [row, column] as a part of the solution and recursively check if placing queens here leads to a solution.

b) If placing queen in [row, column] leads to a solution then return True. else unmark this [row, column] and go to step (a) to try other rows.

iv) If all rows have been tried and nothing worked return false to trigger backtracking.

Program Listing :-

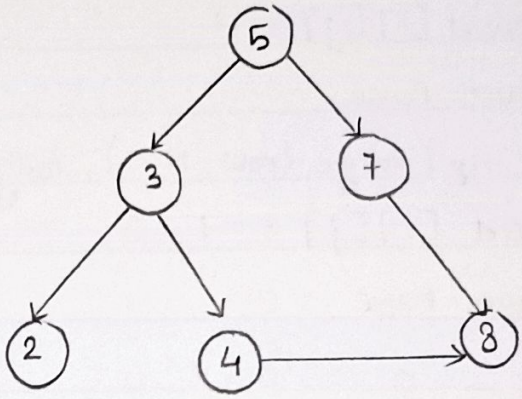
```
global N
N = 4
def print solution (board):
    for i in range (N):
        for j in range (N):
            print (board [i] [j], end = " ")
        print()
def is_safe (board, row, col):
    for i in range (col):
```

PIONEER®

Output :-

```
0  0  1  0
1  0  0  0
0  0  0  1
0  1  0  0
```

```
        if board[row][i] == 1;
            return false
    for i,j in tip(range(row,-1,-1), range(col,-1,-1)):
        if board[i][j] == 1
            return False
    for i,j in tip(range(row, N, 1), range(col, -1,-1)):
        if board[i][j] == 1
            return False
    return True.
def solveNQutil(board, col):
    if col >= N :
        return True
    for i in range(N)
        if is-safe(board, i, col):
            board[i][col] = 1
            if solveNQutil(board, col+1) == True :
                return True
            board[i][col] = 0.
                return False
def Solve-NQ():
    board [ [0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]
    if solveNQutil(board, 0) == False :
        print("solution doesnot exist")
        return False
```



Output :-

5 3 2 4 8 7

DATE

EXPT. NO. 19

PAGE NO.

Depth Limited Search

Algorithm :-

i) For each given nodes in graph G, set status = 1 (ready)

ii) Push the source node into stack and set status = 2 (waiting)

iii) Repeat steps (iv) and (v) until stack is empty or the goal node has been reached.

iv) Pop the top node T of the stack and set status = 3 (visited)

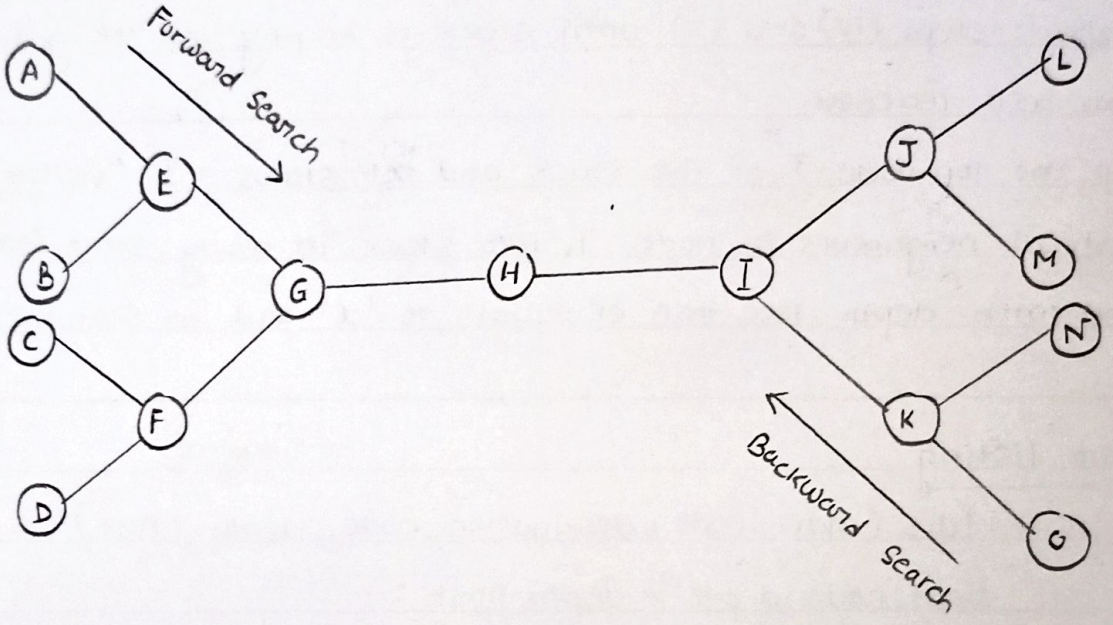
v) Push all neighbours to node T into stack in ready state (status = 1) and with depth less than or equal to 'd' and set status = 2 (wait)

Program listing

```
def graph-ldfs (start_node, destination_node, depth-limit)
        if start_node.depth > depth limit :
                return 0
        start-node . visited = True
        for x in start_node.children :
                if destination_node . visited :
                        break
                if not x.visited :
                        x.parent = start_node
                        x. depth = start_node.depth + 1
                        graph-ldfs (x, destination_node, depth_limit)
```

[As same as DFS].

PIONEER®



Bi-directional Search Algorithm

DATE

EXPT. NO. 20

PAGE NO.

Bi-Directional Search

Algorithm :-

i) A is initial node, O is goal node and H is intersection.

ii) Start searching simultaneously from start to goal node and backward from goal to start nodes.

iii) Whenever the forward search and backward search intersect at a node, the search stops.

Program Listing :-

```
class AdjacentNode :
        def __init__ (self, vertex) :
                self. vertex = vertex
                self. next = None
class BidirectionalSearch :
        def __init__ (self, vertices) :
                self. vertices = vertices
                self. graph = [none] * self. vertices
                self. src_queue = list ( )
                self. dust_queue = list ( )
                self. src_visited = [False] * self. vertices
                self. dust_visited = [False] * self. vertices
                self. src. parent = [None] * self. vertices
                Self. dust_parent = [None] * self. vertices
        def add_edge (self, src, dest) :
```

PIONEER®

```
        node.next = self.graph[src]
        self.graph[src] = node
        node = AdjacentNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node
    def bfs(self, direction='forward'):
        if direction == 'forward':
            current = self.src_queue.pop(0)
            connected_node = self.graph[current]
            while connected_node:
                vertex = connected_node.vertex
                if not self.src_visited[vertex]:
                    self.src_queue.append(vertex)
                    self.src_visited[vertex] = True
                    self.src_parent[vertex] = current
                connected_node = connected_node.next
        else:
            current = self.dest_queue.pop(0)
            connected_node = self.graph[current]
            while connected_node:
                vertex = connected_node.vertex
                if not self.dest_visited[vertex]:
                    self.dest_queue.append(vertex)
                    self.dest_visited[vertex] = True
                    self.dest_parent[vertex] = current
                connected_node = connected_node.next.
```

DATE

EXPT. NO.

PAGE NO.

```
def is_intersecting(self):
    for i in range(self.vertices):
        if(self.src_visited[i] and self.dust_visited[i]):
            return i
        return -1
def print_path(self, intersecting_node, src, dest):
    path = list()
    path.append(intersecting_node)
    i = intersecting_node
    while i != src
        path.append(self.src_parent[i])
        i = self.src_parent[i]
    path = path[::-1]
    i = intersecting_node
    while i != dest:
        path.append(self.dest_parent[i])
        i = self.dest_parent[i]
        print("*** path ***")
        path = list(map(str, path))
        print(' '.join(path))
def bidirectional_search(self, src, dest):
    self.src_queue.append(src)
    self.src.visited[src] = True
    self.src_parent[src] = -1
    self.dest_queue.append(dest)
```

PIONEER®

Output :-

```
graph.addedge (1,4)
graph.add-edge (2,5)
graph.add-edge (3,5)
graph.add-edge (4,6)
graph.add-edge (5,6)
graph.add-edge (6,7)
graph.add-edge (7,8)
graph.add-edge (8,9)
graph.add-edge (8,10)
graph.add-edge (9,11)
graph.add-edge (9,12)
graph.add-edge (10,13)
graph.add-edge (10,14)

Path exist between 0 & 14

Intersection at:7

*** Path ***

0 4 6 7 8 10 14
```

```
        self.dest_visited[dest] = True
        self.dest_parent[dest] = -1
        while self.src_queue and self.dest_queue:
            self.bfs(direction = 'forward')
            self.bfs(direction = 'backward')
            intersecting_node = self.is_intersecting()
            if intersecting_node != -1
                print(f"path exist between {src} and {dest}")
                print(f"Intersection at : {intersecting_node}")
                self.print_path(intersecting_node, src, dest)
                exit(0)
            return -1
if __name__ = '__main__' :
    n = 15
    src = 0
    dest = 14
    graph = Bidirectional search(n)
    graph.addedge(0,4)
out = graph.bidirectional_search(src, dest)
if out == 1
    print(f"Path doesnot exist between {src} and {dest}")
```