Code:

```
from collections import defaultdict
jug1=int(input('Enter the max capacity for jug1 : '))
jug2=int(input('Enter the max capacity for jug1 : '))
aim=int(input('Enter the amount of water to be measured :  '))
visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
        if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
                print(amt1, amt2)
                return True
        if visited[(amt1, amt2)] == False:
                print(amt1, amt2)
                visited[(amt1, amt2)] = True
                return (waterJugSolver(0, amt2) or
                                waterJugSolver(amt1, 0) or
                                waterJugSolver(jug1, amt2) or
                                waterJugSolver(amt1, jug2) or
                                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                                amt2 - min(amt2, (jug1-amt1))) or
                                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                                amt2 + min(amt1, (jug2-amt2))))
        else:
                return False

print("Steps: ")
waterJugSolver(0, 0)
```

Output :

```
Enter the max capacity for jug1 : 4
Enter the max capacity for jug1 : 3
Enter the amount of water to be measured :  2
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
True
```

```
from queue import PriorityQueue
def best_first_search(v,l,goal,h):
  open_list=PriorityQueue()
  open_list.put((10,0))
  flag=0
  path=[]
  while not(open_list.empty()) and flag==0:
    t=open_list.get()
    curr=t[1]
    path.append(curr)
    if(curr==goal):
      flag=1
      break
    for i in range(v):
      if l[curr][i]==1:
        open_list.put((h[i],i))
  if flag==1:
   print('Goal Node is found .The Path is : ',path)
  else :
   print('Goal Node is not found')

v=int(input('Enter the number of Nodes : '))
l=[[0 for i in range(v)] for j in range(v)]
edge=int(input('Enter no of edges : '))
for i in range(edge):
  print('Enter the start ,end of edge-',i+1)
  start, end=[int(x) for x in input().split()]
  l[start][end]=1
print('Adjacency matrix : ')
for i in range(v):
  for j in range(v):
    print(l[i][j],end=' ')
  print()
h={}
for i in range(v):
  print('Enter h(n) for Node',i+1)
  x=int(input())
  h[i]=x
goal=int(input('Enter the goal node :'))
best_first_search(v,l,goal,h)
```

```
Output :
Enter the number of Nodes : 5
Enter no of edges : 4
Enter the start ,end of edge- 1
0 1
Enter the start ,end of edge- 2
0 2
Enter the start ,end of edge- 3
1 3
```

```
Enter the start ,end of edge- 4
1 4
Adjacency matrix :
0 1 1 0 0
0 0 0 1 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Enter h(n) for Node 1
10
Enter h(n) for Node 2
12
Enter h(n) for Node 3
14
Enter h(n) for Node 4
11
Enter h(n) for Node 5
4
Enter the goal node :3
Goal Node is found .The Path is :  [0, 1, 4, 3]
```

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
g = Graph()
v=int(input('Enter no of vertices : '))
e=int(input('Enter no of edges : '))
print('Enter start and end of each edge : ')
for i in range(e) :
  x, y=[int(x) for x in input().split()]
  g.addEdge(x, y)
print("Following is the order of visiting nodes for DFS from (starting from vertex
2) : ")
g.DFS(2)
```

```
Output :
Enter no of vertices : 4
Enter no of edges : 6
Enter start and end of each edge :
0 1
0 2
1 2
2 0
2 3
3 3
Following is the order of visiting nodes for DFS from (starting from vertex 2) :
2 0 1 3
```

```
from collections import defaultdict
class Graph:
  def __init__(self,vertices):
    self.V = vertices
    self.graph = defaultdict(list)
  def addEdge(self,u,v):
    self.graph[u].append(v)
  def DFS(self,src,target,maxDepth):
    print(src,end=' ')
    if src == target : return True
    if maxDepth <= 0 : return False
    for i in self.graph[src]:
        if(self.DFS(i,target,maxDepth-1)):
          return True
    return False
  def DLS(self,src, target, maxDepth):
      if (self.DFS(src, target,maxDepth )):
        return True
      return False
print("Enter number of Vertices : ")
ver = int(input())
print("Enter number of edges : ")
e = int(input())
g = Graph (ver);
print('Enter the edges : ')
for i in range(e):
  x,y = [int(x) for x in (input().split())]
  g.addEdge(x,y)
#target = 6; maxDepth = 3; src = 0
target=int(input('Enter the goal node : '))
maxDepth=int(input('Enter the maximum depth : '))
source=int(input('Enter the source node : '))
print('Path : ')
if g.DLS(source, target, maxDepth) == True:
  print()
  print ("Target is reachable from source " +
    "within max depth")
else :
  print()
  print ("Target is NOT reachable from source " +
    "within max depth")

Output :
Enter number of Vertices :
7
Enter number of edges :
6
Enter the edges :
0 1
0 2
```

```
1 3
1 4
2 5
2 6
Enter the goal node : 6
Enter the maximum depth : 2
Enter the source node : 0
Path :
0 1 3 4 2 5 6
Target is reachable from source within max depth
```

```python
from queue import PriorityQueue
def a_star(v,l,goal,h):
  open_list=PriorityQueue()
  open_list.put((13,0))
  flag=0
  path=[]
  while not(open_list.empty()) and flag==0:
    t=open_list.get()
    curr_cost=t[0]
    curr=t[1]
    path.append(curr)
    if(curr==goal):
      flag=1
      break
    for i in range(v):
      if l[curr][i]!=-1:
        cost=l[curr][i]
        open_list.put((curr_cost-h[curr]+cost+h[i],i))
  if flag==1:
   print('Goal Node is found .The Path is : ',path)
  else :
   print('Goal Node is not found')

v=int(input('Enter the number of Nodes : '))
l=[[-1 for i in range(v)] for j in range(v)]
edge=int(input('Enter no of edges : '))
for i in range(edge):
  print('Enter the start ,end , g(n) (Cost) of edge -',i+1)
  start, end, cost=[int(x) for x in input().split()]
  l[start][end]=cost
print('Adjacency matrix : ')
for i in range(v):
  for j in range(v):
    print(l[i][j],end=' ')
  print()
h={}
for i in range(v):
  print('Enter h(n) for Node',i+1)
  x=int(input())
  h[i]=x
goal=int(input('Enter the goal node :'))
a_star(v,l,goal,h)
```

Output -

```
Enter the number of Nodes : 10
Enter no of edges : 9
Enter the start ,end , g(n) (Cost) of edge - 1
0 1 3
Enter the start ,end , g(n) (Cost) of edge - 2
0 2 2
```

```
Enter the start ,end , g(n) (Cost) of edge - 3
1 3 4
Enter the start ,end , g(n) (Cost) of edge - 4
1 4 1
Enter the start ,end , g(n) (Cost) of edge - 5
2 5 3
Enter the start ,end , g(n) (Cost) of edge - 6
2 6 1
Enter the start ,end , g(n) (Cost) of edge - 7
5 7 5
Enter the start ,end , g(n) (Cost) of edge - 8
6 8 2
Enter the start ,end , g(n) (Cost) of edge - 9
6 9 3
Adjacency matrix :
-1 3 2 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 4 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 3 1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 5 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 2 3
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Enter h(n) for Node 1
13
Enter h(n) for Node 2
12
Enter h(n) for Node 3
4
Enter h(n) for Node 4
7
Enter h(n) for Node 5
3
Enter h(n) for Node 6
8
Enter h(n) for Node 7
2
Enter h(n) for Node 8
4
Enter h(n) for Node 9
9
Enter h(n) for Node 10
0
Enter the goal node :7
Goal Node is found .The Path is :  [0, 2, 6, 9, 5, 7]
```

```
def isSafe(mat, r, c):
    for i in range(r):
        if mat[i][c] == 'Q':
            return False
    (i, j) = (r, c)
    while i >= 0 and j >= 0:
        if mat[i][j] == 'Q':
            return False
        i = i - 1
        j = j - 1
    (i, j) = (r, c)
    while i >= 0 and j < len(mat):
        if mat[i][j] == 'Q':
            return False
        i = i - 1
        j = j + 1
    return True

def printSolution(mat):
    for r in mat:
        print(str(r).replace(',', '').replace('\'', ''))
    print()

def nQueen(mat, r):
    if r == len(mat):
        printSolution(mat)
        return
    for i in range(len(mat)):
        if isSafe(mat, r, i):
            mat[r][i] = 'Q'
            nQueen(mat, r + 1)
            mat[r][i] = '-'
N = 4
mat = [['-' for x in range(N)] for y in range(N)]
print('Possible Solutions : ')
nQueen(mat, 0)
```

Output:

```
Possible Solutions :
[- Q - -]
[- - - Q]
[Q - - -]
[- - Q -]

[- - Q -]
[Q - - -]
[- - - Q]
[- Q - -]
```

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
    def isSafe(self, v, colour, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and colour[i] == c:
                return False
        return True
    def graphColourUtil(self, m, colour, v):
        if v == self.V:
            return True
        for c in range(1, m + 1):
            if self.isSafe(v, colour, c) == True:
                colour[v] = c
                if self.graphColourUtil(m, colour, v + 1) == True:
                    return True
                colour[v] = 0
    def graphColouring(self, m):
        colour = [0] * self.V
        if self.graphColourUtil(m, colour, 0) == None:
            return False
        print("Solution exist and Following are the assigned colours:")
        for c in colour:
            print(c, end=' ')
v=int(input('Enter number of vertices : '))
g = Graph(v)
print('Enter the adjacency matrix : ')
g.graph = [[int(x) for x in input().split()] for b in range(v)]
m = int(input('Enter no of colours : '))
g.graphColouring(m)
```