

# **DELHI TECHNOLOGICAL UNIVERSITY**

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi 110042

## **DEPARTMENT OF SOFTWARE ENGINEERING**



**SE321: Artificial Intelligence**

**A2- G4**

**Lab File**

**Submitted To:**

**Ms. Anjali Bansal  
Department of Software  
Engineering**

**Submitted By:**

***Summerprit singh  
2K19/SE/129***

## Experiment 1

**Aim:** To find the Depth first search traversal and breadth first search traversal of a graph.

### Theory:

**Depth first search (DFS)** algorithm starts with the initial node of the graph  $G$ , and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack/recursion stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

**Breadth-first search (BFS)** is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

### Algorithm:

```
DFS(G, u)

    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
    For each u ∈ G
        DFS(G, u)
}
```

```
create a queue Q

mark v as visited and put v into Q

while Q is non-empty

    remove the head u of Q

    mark and enqueue all (unvisited) neighbours of u
```

### Source Code:

```
def dfs(src,g,vis):
    print(src,end=" ")
    vis[src]=1
    for nbrs in g[src]:
        if not vis[nbrs]:
            dfs(nbrs,g,vis)

def bfs(g,nodes):
    q=[]
    q.append(0)
    vis=[]
    for i in range(0, nodes):
        vis.append(0)
    while(len(q)):
        node=q.pop(0)
        if vis[node]:
            continue
        print(node,end=" ")
        vis[node]=1
        for nbr in g[node]:
            if not vis[nbr]:
                q.append(nbr)

def main():
    nodes=int(input("Number of nodes "))
    edges=int(input("Number of edges "))
    vis=[]
    g=[]
    for i in range(0,nodes):
        vis.append(0)
        g.append([])
    print("Add Edges (nodes are numbered from 0 to n-1)")
    for i in range(0,edges):
        src,des=map(int,input().split())
        g[src].append(des)
```

```

        g[des].append(src)

    print("DFS of graph:",end=" ")
    dfs(0,g,vis)
    print()
    print("BFS of graph:",end=" ")
    bfs(g,nodes)

main()

```

## Result:

Jupyter Notebook interface showing the execution of a graph algorithm. The code defines a graph with 8 nodes and 9 edges, then performs DFS and BFS traversals. The output shows the number of nodes and edges, the edges themselves, and the traversal paths for both DFS and BFS.

```

main()

Number of nodes 8
Number of edges 9
Add Edges (nodes are numbered from 0 to n-1)
0 1
0 2
1 5
1 3
2 3
2 4
3 4
3 6
4 7
DFS of graph: 0 1 5 3 2 4 7 6
BFS of graph: 0 1 2 5 3 4 6 7

```

## Conclusion:

DFS can be used to find path between two nodes, check if graphs are bipartite, detect cycle, Topological sorting etc. And Time complexity of DFS is  $O(V+E)$  where  $V$  is number of nodes and  $E$  is number of edges.

BFS is used to find smallest bath between the two nodes in terms of edges, serialize and deserialize the binary tree, for computing the maximum flow in a flow network. And Time complexity of BFS is  $O(V+E)$  where  $V$  is number of nodes and  $E$  is number of edges.

## Experiment 2

**Aim:** To find the factorial of a given number.

**Theory:** Factorial, in mathematics is the product of all positive integers less than or equal to a given positive integer and denoted by that integer and an exclamation point. Thus, factorial seven is written 7! meaning  $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$ . Factorial zero is defined as equal to 1.

### Algorithm:

```
def fac(number)


    for value = 1 to number
        factorial = factorial * value
    end for
    return factorial

end fac
```

### Source Code:

```
def fac(n):
    ans=1
    for i in range(2,n+1):
        ans=ans*i
    return ans
def main():
    n=int(input("Enter number to find factorial "))
    if n<0:
        print("INVALID INPUT!!")
    else:
        print("Factorial of number is:",end=" ")
        print(fac(n))
main()
```

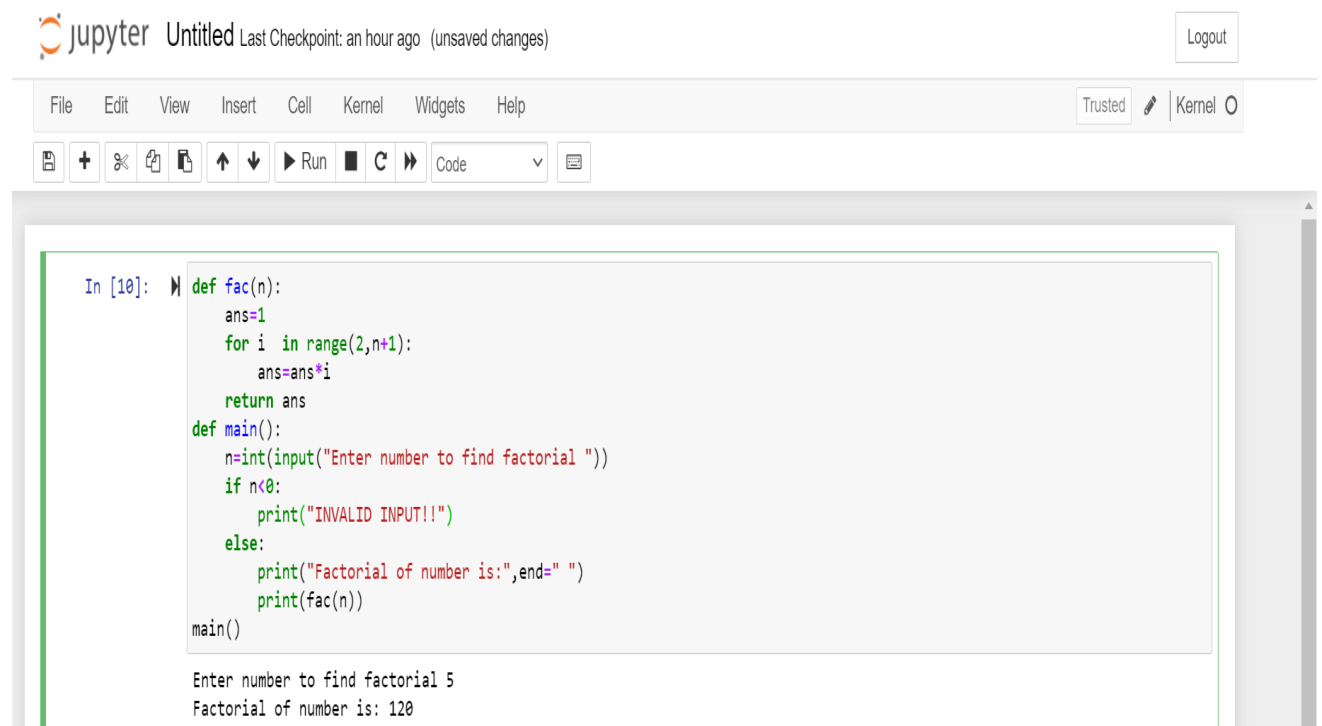
## Result:



A screenshot of a Jupyter Notebook interface. The top bar shows the Jupyter logo, the text "Untitled", and "Last Checkpoint: an hour ago (autosaved)". A "Logout" button is in the top right. Below the top bar is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". To the right of the menu bar are "Trusted" and "Kernel" buttons. Below the menu bar is a toolbar with icons for saving, adding, deleting, and running code. The main area contains a code cell with the following Python code:

```
In [9]: def fac(n):
        ans=1
        for i in range(2,n+1):
            ans=ans*i
        return ans
def main():
    n=int(input("Enter number to find factorial "))
    if n<0:
        print("INVALID INPUT!!")
    else:
        print("Factorial of number is:",end=" ")
        print(fac(n))
main()
```

The output of the code cell shows the prompt "Enter number to find factorial -8" followed by "INVALID INPUT!!".



A screenshot of a Jupyter Notebook interface, similar to the one above. The top bar shows "Untitled" and "Last Checkpoint: an hour ago (unsaved changes)". The "Logout" button is in the top right. The menu bar and toolbar are the same. The code cell contains the same Python code as in the previous screenshot:

```
In [10]: def fac(n):
        ans=1
        for i in range(2,n+1):
            ans=ans*i
        return ans
def main():
    n=int(input("Enter number to find factorial "))
    if n<0:
        print("INVALID INPUT!!")
    else:
        print("Factorial of number is:",end=" ")
        print(fac(n))
main()
```

The output of the code cell shows the prompt "Enter number to find factorial 5" followed by "Factorial of number is: 120".

**Conclusion:** Factorials are commonly encountered in the evaluation of permutation and combination and in the coefficients of terms of binomial expansions.

## Experiment 3

**Aim:** Write a program for the Monkey banana problem.

**Theory:** A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a block could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air.

### Algorithm:

Step 1. Take the positions of monkey, block and banana inside the room.

Step 2. Marks these points in the grid as 1 for monkey 2 for block and 3 for banana.

Step 3. Apply BFS (Breadth first search) by passing source position as monkey's initial position and target position as block's initial position. Implementation of BFS is Explained in 1<sup>st</sup> Experiment.

Step 4. From step 3 we will get path in which monkey would travel to reach the block.

Step 5. Now we will again apply BFS by passing monkey's current position as a source and target position as banana's position.

Step 6. From step 5 we will get path in which monkey would travel to get bananas.

### Source Code:

```
class item:
    def __init__(self,i,j,psf):
        self.i=i
        self.j=j
        self.psf=psf

def isValid(x,y,n,m,vis):
    if(x<0 or y<0 or x>=n or y>=m or vis[x][y]==1):
        return False
    return True

def bfs(grid,posMonkey,n,m,target):
    x=[0,1,0,-1,1,-1,1,-1]
    y=[1,0,-1,0,1,-1,-1,1]
    vis=[[0 for j in range(0,m)] for i in range(0,n)]
    q=[]
    q.append(item(posMonkey[0],posMonkey[1],""))
    vis[posMonkey[0]][posMonkey[1]]=1
```

```

while(len(q)>0):
    rem=q.pop(0)
    i=rem.i
    j=rem.j
    if(grid[i][j]==target):
        return rem.psf
    for k in range(0,8):
        if(isValid(i+x[k],j+y[k],n,m,vis)):
            vis[i+x[k]][j+y[k]]=1
            if (k == 0):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"R"))
            if (k == 1):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"D"))
            if (k == 2):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"L"))
            if (k == 3):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"U"))
            if (k == 4):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"DDR"))
            if (k == 5):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"DUL "))
            if (k == 6):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"DDL"))
            if (k == 7):
                q.append(item(i + x[k], j + y[k], rem.psf+" "+"DUR"))

def main():
    n,m=map(int,input("SIZE OF A ROOM: ").split())
    grid=[[0 for j in range(0,m)] for i in range(0,n)]
    posMonkey=list(map(int,input("Enter position of monkey: ").split())) #1
    posBlock =list(map(int, input("Enter position of block: ").split())) #2
    posBanana =list(map(int, input("Enter position of banana: ").split())) #3
    print()
    try:
        grid[posMonkey[0]][posMonkey[1]]=1
        grid[posBlock[0]][posBlock[1]] = 2
        grid[posBanana[0]][posBanana[1]]=3
    except LookupError:
        print ("Invalid input!!! location out of the room")
    else:

        print("0-->Initial Empty spaces positon")
        print("1-->Initial Monkey position")
        print("2-->Initial Block position")
        print("3-->Initial Banana position")
        print()

```



```

for i in range(0,n):
    for j in range(0,m):
        print(grid[i][j],end=" ")
    print()
print()
print("Movement of Monkey to get Bananas as soon as Possible")
print()
if((posBlock==posBanana) and (posBanana==posMonkey)):
    print("Climb the Block and get Bananas")
elif(posMonkey==posBlock):
    print("Grab the Block")
    monkeyToBanana=bfs(grid,posMonkey,n,m,3)
    bbm=monkeyToBanana.split()

    for i in bbm:
        if(i=="R"):
            print("Move Right")
        if(i=="L"):
            print("Move Left")
        if (i == "DDR"):
            print("Move Diagonaly Down Right")
        if (i == "DDL"):
            print("Move Diagonaly Down Left")
        if (i == "DUL"):
            print("Move Diagonaly Up Left")
        if (i == "DUR"):
            print("Move Diagonaly Up Right")
        if (i == "U"):
            print("Move Up")
        if (i == "D"):
            print("Move Down")

    print("Climb the Block and get Bananas")

elif(posBanana==posBlock):

    monkeyToBanana=bfs(grid,posMonkey,n,m,3)
    bbm=monkeyToBanana.split()

    for i in bbm:
        if(i=="R"):
            print("Move Right")
        if(i=="L"):
            print("Move Left")
        if (i == "DDR"):
            print("Move Diagonaly Down Right")
        if (i == "DDL"):
            print("Move Diagonaly Down Left")

```

```

        if (i == "DUL"):
            print("Move Diagonaly Up Left")
        if (i == "DUR"):
            print("Move Diagonaly Up Right")
        if (i == "U"):
            print("Move Up")
        if (i == "D"):
            print("Move Down")

        print("Climb the Block and get Bananas")

    else:
        monkeyToBlock=bfs(grid,posMonkey,n,m,2)
        print()
        mbm=monkeyToBlock.split()
        for i in mbm:
            if(i=="R"):
                print("Move Right")
            if(i=="L"):
                print("Move Left")
            if (i == "DDR"):
                print("Move Diagonaly Down Right")
            if (i == "DDL"):
                print("Move Diagonaly Down Left")
            if (i == "DUL"):
                print("Move Diagonaly Up Left")
            if (i == "DUR"):
                print("Move Diagonaly Up Right")
            if (i == "U"):
                print("Move Up")
            if (i == "D"):
                print("Move Down")

        print("Grab the Block")

        monkeyToBanana=bfs(grid,posBlock,n,m,3)
        bbm=monkeyToBanana.split()

        for i in bbm:
            if(i=="R"):
                print("Move Right")
            if(i=="L"):
                print("Move Left")
            if (i == "DDR"):
                print("Move Diagonaly Down Right")
            if (i == "DDL"):
                print("Move Diagonaly Down Left")
            if (i == "DUL"):

```

```
        print("Move Diagonaly Up Left")
    if (i == "DUR"):
        print("Move Diagonaly Up Right")
    if (i == "U"):
        print("Move Up")
    if (i == "D"):
        print("Move Down")

    print("Climb the Block and get Bananas")
```

```
main()
```

## Result:

---

SIZE OF A ROOM: 4 4

Enter position of monkey: 2 2

Enter position of block: 1 1

Enter position of banana: 3 3

0-->Initial Empty spaces position

1-->Initial Monkey position

2-->Initial Block position

3-->Initial Banana position

0 0 0 0

0 2 0 0

0 0 1 0

0 0 0 3

Movement of Monkey to get Bananas as soon as Possible

Move Diagonally Up Left

Grab the Block

Move Diagonally Down Right

Move Diagonally Down Right

Climb the Block and get Bananas

---

SIZE OF A ROOM: 10 10  
Enter position of monkey: 0 0  
Enter position of block: 2 9  
Enter position of banana: 5 5

0-->Initial Empty spaces positon  
1-->Initial Monkey position  
2-->Initial Block position  
3-->Initial Banana position

```
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 3 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Movement of Monkey to get Bananas as soon as Possible

Move Right  
Move Right  
Move Right  
Move Right  
Move Right  
Move Right  
Move Right  
Move Diagonaly Down Right  
Move Diagonaly Down Right  
Grab the Block  
Move Left  
Move Diagonaly Down Left  
Move Diagonaly Down Left  
Move Diagonaly Down Left  
Climb the Block and get Bananas

---

**Conclusion:** Monkey and Banana problem is implemented successfully using breadth first search to reach the target with minimum number of steps. And Time complexity of code is  $O(N*M)$  where N, M are dimensions of room.

## Experiment 4

**Aim:** Write a program for the Water Jug problem.

**Theory:** Given two jugs with the maximum capacity of  $m$  and  $n$  liters respectively. The jugs don't have markings on them which can help us to measure smaller quantities. The task is to measure  $d$  liters of water using these two jugs. Hence our goal is to reach from initial state  $(m, n)$  to final state  $(0, d)$  or  $(d, 0)$ .

At any point, there can be a total of six possibilities:

1. Empty the first jug completely
2. Empty the second jug completely
3. Fill the first jug
4. Fill the second jug
5. Fill the water from the second jug into the first jug until the first jug is full or the second jug has no water left
6. Fill the water from the first jug into the second jug until the second jug is full or the first jug has no water left

**Algorithm:**

1. We can use Recursion for the above problem.
2. Visit all the six possible moves one by one until one of them returns True.
3. Since there can be repetitions of same recursive calls, hence every return value is stored using memoization to avoid calling the recursive function again and returning the stored value.

**Source Code:**

```
from collections import defaultdict
jug1, jug2, target = map(int, input("Enter capacities of Jug1 and jug 2, and Target: ").split())

visited = defaultdict(lambda: False)

def waterJug(currCap1, currCap2):

    if (currCap1 == target and currCap2 == 0) or (currCap2 == target and currCap1 == 0):
        print(currCap1, currCap2)
        return True

    if visited[(currCap1, currCap2)] == False:
        print(currCap1, currCap2)

        visited[(currCap1, currCap2)] = True
```

```

        return (waterJugSolver(0, currCap2) or
                waterJugSolver(currCap1, 0) or
                waterJugSolver(jug1, currCap2) or
                waterJugSolver(currCap1, jug2) or
                waterJugSolver(currCap1 + min(currCap2, (jug1-currCap1)),
                                currCap2 - min(currCap2, (jug1-currCap1))) or
                waterJugSolver(currCap1 - min(currCap1, (jug2-currCap2)),
                                currCap2 + min(currCap1, (jug2-currCap2))))

    else:
        return False

print("Sequence of Steps: ")
waterJug(0, 0)

```

### Output:

1. When solution is not possible.

```

Enter capacities of Jug1 and jug 2, and Target: 4 5 3
Sequence of Steps:

```

```

0 0
4 0
4 5
0 5
4 1
0 1
1 0
1 5
4 2
0 2
2 0
2 5
4 3
0 3
3 0
3 5
4 4
0 4

```

Out[21]: False



2. When solution is possible.

```
Enter capacities of Jug1 and jug 2, and Target: 4 3 2
```

```
Sequence of Steps:
```

```
0 0
```

```
4 0
```

```
4 3
```

```
0 3
```

```
3 0
```

```
3 3
```

```
4 2
```

```
0 2
```

**Out[4]:** True

**Conclusion:** Water Jug problem is implemented successfully using depth first search. The space complexity is  $O(M*N)$  and time complexity of the code is  $O(M*N)$

## Experiment 5

**Aim:** Write a program for tic tac toe problem.

**Theory:** Tic Tac Toe is two player Game which is played on 3X3 dimension board. Players put their selected symbol “X” or “O” at empty positions of the board. To win the game, a player must place three of their marks in a horizontal, vertical, or diagonal row.

The following example game is won by the first player, X:



### Algorithm:

1. Take intermediate input of 3X3 Tac Tac Toe board.
2. Analyse the empty positions.
3. Using mini-max try to find best move which can be made.
4. Fill the empty position with best move with player's selected symbol (“x” or “o”).
5. These four steps will be repeating to get further optimal moves.

### Source Code:

```
player, opponent = 'x', 'o'

def isMovesLeft(board) :

    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False

def evaluate(b) :

    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            elif (b[row][0] == opponent) :
                return -10

    for col in range(3) :

        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

            if (b[0][col] == player) :
                return 10
```



```

else :
    best = 1000

    for i in range(3) :
        for j in range(3) :

            if (board[i][j] == '_' ) :

                board[i][j] = opponent

                best = min(best, minimax(board, depth + 1, not isMax))

                board[i][j] = '_'
    return best

def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    for i in range(3) :
        for j in range(3) :

            if (board[i][j] == '_' ) :

                board[i][j] = player

                moveVal = minimax(board, 0, False)

                board[i][j] = '_'

                if (moveVal > bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal

    print()
    return bestMove

def main():

    board=[[0 for i in range(0,3)] for j in range(0,3)]

    print("Given matrix should be of 3x3 dimension")
    print()

```

```

    print("Enter a board of tic tac toe when nor player 1 and neither player 2
is won to get optimal move.")

    for i in range(0,3):
        board[i]=list(input("Enter values of row{:}.format(i)).split())

    print()
    print("The given board is: ")
    for i in range(0,3):
        print(board[i])
    bestMove = findBestMove(board)

    print("The Optimal Move is :")
    print("ROW:", bestMove[0], " COL:", bestMove[1])

main()

```

## Output:

Given matrix should be of 3x3 dimension

Enter a board of tic tac toe when nor player 1 and neither player 2 is won to get optimal mo

Enter values of row0: x x \_

Enter values of row1: o \_ \_

Enter values of row2: o \_ \_

The given board is:

['x', 'x', '\_']

['o', '\_', '\_']

['o', '\_', '\_']

The Optimal Move is :

ROW: 0 COL: 2

Given matrix should be of 3x3 dimension

Enter a board of tic tac toe when nor player 1 and neither player 2 is won to get optimal move.

Enter values of row0: x o x

Enter values of row1: o x o

Enter values of row2: \_ \_ \_

The given board is:

['x', 'o', 'x']

['o', 'x', 'o']

['\_', '\_', '\_']

The Optimal Move is :

ROW: 2 COL: 0

---

Given matrix should be of 3x3 dimension

Enter a board of tic tac toe when nor player 1 and neither player 2 is won to get optimal move.

Enter values of row0: o \_ o

Enter values of row1: x x \_

Enter values of row2: \_ x o

The given board is:

['o', '\_', 'o']

['x', 'x', '\_']

['\_', 'x', 'o']

The Optimal Move is :

ROW: 0 COL: 1

**Conclusion:** AI solution for tic tac toe is implemented successfully using Mini-Max algorithm to decide the next optimal move. And Time complexity of code is  $O(1)$  because given matrix is of 3X3 means input size is always constant.