



# ARTIFICIAL INTELLIGENCE LAB MANUAL

B.Tech (CSE) – R19 & R20

**PROF. T. KAMESWARA RAO**

Professor, Dept. of CSE

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

NAMBURU – GUNTUR DISTRICT - AP



## VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY (Autonomous)

Permanently Affiliated to JNTU Kakinada, Approved by AICTE  
Accredited by NAAC with 'A' Grade, ISO 9001:2008 Certified  
Nambur, Pedakakani (M), Guntur (Dt) - 522508

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**  
B.Tech Program is Accredited by NBA

Vision of the Department	
To facilitate quality education by focusing on assimilation, generation and dissemination of knowledge in the area of Computer Science & Engineering to transform students into socially responsible engineers.	
Mission of the Department	
<ul style="list-style-type: none"> <li>Equip our graduates with the knowledge by student centric teaching-learning process and expertise to contribute significantly to the software industry and to continue to grow professionally.</li> <li>To train socially responsible, disciplined engineers who work with good leadership skills and can contribute for nation building.</li> <li>To make our graduates aware of cutting edge technologies and make them industry-ready engineers.</li> <li>To shape the department into a Centre of Academic and Research excellence.</li> </ul>	

Program Educational Objectives	
PEO-1	To provide the graduates with <i>solid foundation in Computer Science and Engineering</i> along with the fundamentals of Mathematics and Sciences with a view to impart in them high quality technical skills like modelling, analysing, designing, programming and implementation with global competence and helps the graduates for life-long learning.
PEO-2	To prepare and motivate graduates with <i>recent technological developments related to core subjects</i> like Programming, Databases, Design of Compilers and Network Security aspects and future technologies so as to contribute effectively for Research & Development by participating in professional activities like publishing and seeking copy rights.
PEO-3	To train graduates to choose a <i>decent career option either in high degree of employability/Entrepreneur or, in higher education</i> by empowering students with ethical administrative acumen, ability to handle critical situations and training to excel in competitive examinations.
PEO-4	To train the graduates to have <i>basic interpersonal skills and sense of social responsibility</i> that paves them a way to become good team members and leaders.

**Artificial Intelligence Lab Course details**

1. Course : B.Tech
2. Branch : CSE
3. Regulation : R19 & R20
4. Code : 20CS5L02
5. Lab Name : Artificial Intelligence Lab
6. Total Marks : 75 (Internal Lab marks 25 + External lab marks 50)

**Course Objectives:**

1. Study the concepts of Artificial Intelligence.
2. Learn the methods of solving problems using Artificial Intelligence.
3. Introduce the concepts of machine learning.

**Course Outcomes:**

At the end of the course, the students will be able to:

**CO1:** Identify problems that are amenable to solution by AI methods.

**CO2:** Recognize appropriate AI methods to solve a given problem.

**CO3:** Discuss a given problem in the framework of different AI methods.

**CO4:** Develop basic AI algorithms

**Pre Requisites:**

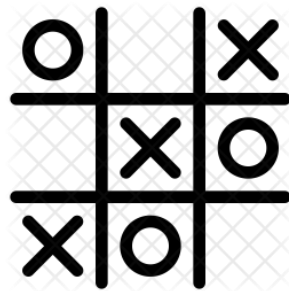
- Knowledge in Python programming
- Knowledge in Algorithms

## Contents

S. No.	Experiment	P. No.
1.	Write a Program to Implement Tic-Tac-Toe game	
2.	Write a program to solve Water-Jug problem	
3.	Write a Program to Implement Breadth First Search	
4.	Write a Program to Implement Depth First Search	
5.	Write a Program to Implement 8-Puzzle problem	
6.	Implementation of Towers of Hanoi Problem	
7.	Write a Program to Implement Missionaries-Cannibals Problem	
8.	Write a Program to Implement Travelling Salesman Problem	
9.	Write a Program to Implement N-Queens Problem	

## 1. Write a Program to Implement Tic-Tac-Toe game

Tic-Tac-Toe is a simple game for two players that we enjoyed playing as kids (especially in boring classrooms). The game involves 2 players placing their respective symbols in a 3x3 grid. The player who manages to place three of their symbols in horizontal/vertical/diagonal row wins the game. If either player fails to do so the game ends in a draw. If both the people always play their optimal strategies the game always ends in a draw.



Ex: Tic-Tac-Toe Board

### Python Code for Tic Tac Toe

```
import random

def drawBoard(board):
    # This function prints out the board that it was passed.
    # "board" is a list of 10 strings representing the board (ignore index 0)
    print(' | | ')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print(' | | ')
    print('-----')
    print(' | | ')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print(' | | ')
    print('-----')
    print(' | | ')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print(' | | ')
```

```

def inputPlayerLetter():
    # Lets the player type which letter they want to be.
    # Returns a list with the player's letter as the first item, and the computer's letter as
    the second.
    letter = ''
    while not (letter == 'X' or letter == 'O'):
        print('Do you want to be X or O?')
        letter = input().upper()
    # the first element in the tuple is the player's letter, the second is the computer's
    letter.
    if letter == 'X':
        return ['X', 'O']
    else:
        return ['O', 'X']

def whoGoesFirst():
    # Randomly choose the player who goes first.
    if random.randint(0, 1) == 0:
        return 'computer'
    else:
        return 'player'

def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns
    False.
    print('Do you want to play again? (yes or no)')
    return input().lower().startswith('y')

def makeMove(board, letter, move):
    board[move] = letter

def isWinner(bo, le):
    # Given a board and a player's letter, this function returns True if that player has
    won.

```

# We use bo instead of board and le instead of letter so we don't have to type as much.

```

return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the top
(bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
(bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
(bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
(bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
(bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
(bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
(bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal

```

def getBoardCopy(board):

# Make a duplicate of the board list and return it the duplicate.

dupeBoard = []

for i in board:

dupeBoard.append(i)

return dupeBoard

def isSpaceFree(board, move):

# Return true if the passed move is free on the passed board.

return board[move] == ' '

def getPlayerMove(board):

# Let the player type in his move.

move = ' '

while move not in '1 2 3 4 5 6 7 8 9'.split() or not isSpaceFree(board, int(move)):

print('What is your next move? (1-9)')

move = input()

return int(move)

def chooseRandomMoveFromList(board, movesList):

# Returns a valid move from the passed list on the passed board.

# Returns None if there is no valid move.

possibleMoves = []

for i in movesList:

```

    if isSpaceFree(board, i):
        possibleMoves.append(i)
    if len(possibleMoves) != 0:
        return random.choice(possibleMoves)
    else:
        return None

def getComputerMove(board, computerLetter):
    # Given a board and the computer's letter, determine where to move and return that
    move.

    if computerLetter == 'X':
        playerLetter = 'O'
    else:
        playerLetter = 'X'

    # Here is our algorithm for our Tic Tac Toe AI:
    # First, check if we can win in the next move
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, computerLetter, i)
            if isWinner(copy, computerLetter):
                return i

    # Check if the player could win on his next move, and block them.
    for i in range(1, 10):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, playerLetter, i)
            if isWinner(copy, playerLetter):
                return i

    # Try to take one of the corners, if they are free.
    move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move != None:

```



```

        return move
    # Try to take the center, if it is free.
    if isSpaceFree(board, 5):
        return 5
    # Move on one of the sides.
    return chooseRandomMoveFromList(board, [2, 4, 6, 8])
def isBoardFull(board):
    # Return True if every space on the board has been taken. Otherwise return False.
    for i in range(1, 10):
        if isSpaceFree(board, i):
            return False
    return True
print('Welcome to Tic Tac Toe!')
while True:
    # Reset the board
    theBoard = [' '] * 10
    playerLetter, computerLetter = inputPlayerLetter()
    turn = whoGoesFirst()
    print('The ' + turn + ' will go first.')
    gameIsPlaying = True
    while gameIsPlaying:
        if turn == 'player':
            # Player's turn.
            drawBoard(theBoard)
            move = getPlayerMove(theBoard)
            makeMove(theBoard, playerLetter, move)
            if isWinner(theBoard, playerLetter):
                drawBoard(theBoard)
                print('Hooray! You have won the game!')
                gameIsPlaying = False
            else:

```

```

        if isBoardFull(theBoard):
            drawBoard(theBoard)
            print("The game is a tie!")
            break
        else:
            turn = 'computer'
    else:
        # Computer's turn.
        move = getComputerMove(theBoard, computerLetter)
        makeMove(theBoard, computerLetter, move)

    if isWinner(theBoard, computerLetter):
        drawBoard(theBoard)
        print("The computer has beaten you! You lose.")
        gameIsPlaying = False
    else:
        if isBoardFull(theBoard):
            drawBoard(theBoard)
            print("The game is a tie!")
            break
        else:
            turn = 'player'

    if not playAgain():
        break

```

**OUTPUT: -**

```

WELCOME TO TIC TAC TOE!
DO YOU WANT TO BE X OR O?
X
THE PLAYER WILL GO FIRST.
| |
| |

```

| |

-----

| |

| |

| |

-----

| |

| |

| |

WHAT IS YOUR NEXT MOVE? (1-9)

5

| |

| |

| |

-----

| |

| X |

| |

-----

| |

O | |

| |

WHAT IS YOUR NEXT MOVE? (1-9)

3

| |

O | |

| |

-----

| |

| X |

| |

-----

```

  | |
O | | X
  | |

```

WHAT IS YOUR NEXT MOVE? (1-9)

4

```

  | |
O | |
  | |

```

-----

```

  | |
X | X | O
  | |

```

-----

```

  | |
O | | X
  | |

```

WHAT IS YOUR NEXT MOVE? (1-9)

3

WHAT IS YOUR NEXT MOVE? (1-9)

8

```

  | |
O | X |
  | |

```

-----

```

  | |
X | X | O
  | |

```

-----

```

  | |
O | O | X

```

| | |

WHAT IS YOUR NEXT MOVE? (1-9)

3

WHAT IS YOUR NEXT MOVE? (1-9)

2

WHAT IS YOUR NEXT MOVE? (1-9)

9

| | |

O | X | X

| | |

-----

| | |

X | X | O

| | |

-----

| | |

O | O | X

| | |

THE GAME IS A TIE!

DO YOU WANT TO PLAY AGAIN? (YES OR NO)

## 2. Write a program to solve Water-Jug problem

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

Here the initial state is (0, 0). The goal state is (2, n) for any value of n.

**State Space Representation:** we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note that  $0 \leq x \leq 4$ , and  $0 \leq y \leq 3$ .

To solve this we have to make some assumptions not mentioned in the problem. They are:

- We can fill a jug from the pump.
- We can pour water out of a jug to the ground.
- We can pour water from one jug to another.
- There is no measuring device available.

Operators — we must define a set of operators that will take us from one state to another.

Sr.	Current State	Next State	Descriptions
1	$(x, y)$ if $x < 4$	$(4, y)$	Fill the 4 gallon jug
2	$(x, y)$ if $y < 3$	$(x, 3)$	Fill the 3 gallon jug
3	$(x, y)$ if $x > 0$	$(x - d, y)$	Pour some water out of the 4 gallon jug
4	$(x, y)$ if $y > 0$	$(x, y - d)$	Pour some water out of the 3 gallon jug
5	$(x, y)$ if $y > 0$	$(0, y)$	Empty the 4 gallon jug
6	$(x, y)$ if $y > 0$	$(x, 0)$	Empty the 3 gallon jug on the ground
7	$(x, y)$ if $x + y \geq 4$ and $y > 0$	$(4, y - (4 - x))$	Pour water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full
8	$(x, y)$ if $x + y \geq 3$ and $x > 0$	$(x - (3 - y), 3)$	Pour water from the 4 gallon jug into the 3-gallon jug until the 3 gallon jug is full
9	$(x, y)$ if $x + y \leq 4$ and $y > 0$	$(x + y, 0)$	Pour all the water from the 3 gallon jug into the 4 gallon jug
10	$(x, y)$ if $x + y \leq 3$ and $x > 0$	$(0, x + y)$	Pour all the water from the 4 gallon jug into the 3 gallon jug
11	$(0, 2)$	$(2, 0)$	Pour the 2 gallons from 3 gallon jug into the 4 gallon jug
12	$(2, y)$	$(0, y)$	Empty the 2 gallons in the 4 gallon jug on the ground

There are several sequences of operations that will solve the problem.

One of the possible solutions is given as:

Gallons in the 4-gallon jug	Gallons in the 3-gallon jug	Rule applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5 or 12
0	2	9 or 11
2	0	--

**Code:**

```

print("Water Jug Problem")
x=int(input("Enter X:"))
y=int(input("Enter Y:"))
while True:
    rno=int(input("Enter the Rule No"))
    if rno==1:
        if x<4:x=4

    if rno==2:
        if y<3:y=3

    if rno==5:
        if x>0:x=0

    if rno==6:
        if y>0:y=0

    if rno==7:
        if x+y>= 4 and y>0:x,y=4,y-(4-x)

    if rno==8:
        if x+y>=3 and x>0:x,y=x-(3-y),3

    if rno==9:
        if x+y<=4 and y>0:x,y=x+y,0

    if rno==10:
        if x+y<=3 and x>0:x,y=0,x+y

    print("X =" ,x)
    print("Y =" ,y)
    if (x==2):
        print(" The result is a Goal state")
        break

```

**Output:**

```

Water Jug Problem
Enter X:0
Enter Y:0
Enter the Rule No2
X = 0
Y = 3
Enter the Rule No9
X = 3
Y = 0
Enter the Rule No2

```



$X = 3$

$Y = 3$

Enter the Rule No7

$X = 4$

$Y = 2$

Enter the Rule No5

$X = 0$

$Y = 2$

Enter the Rule No9

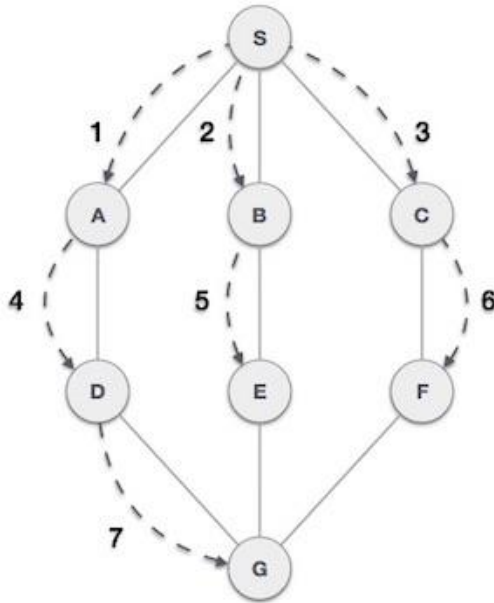
$X = 2$

$Y = 0$

The result is a Goal state.

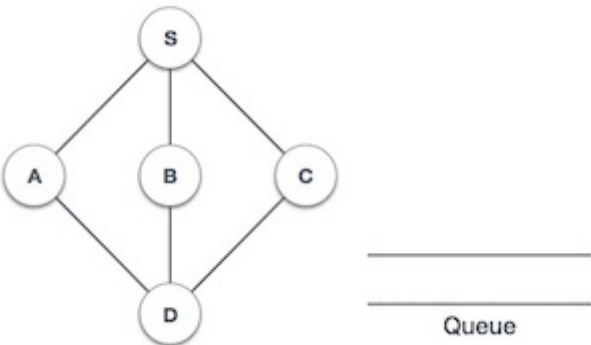
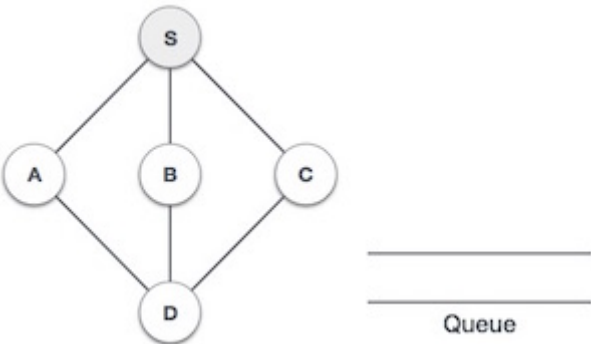
### 3. Write a Program to Implement Breadth First Search

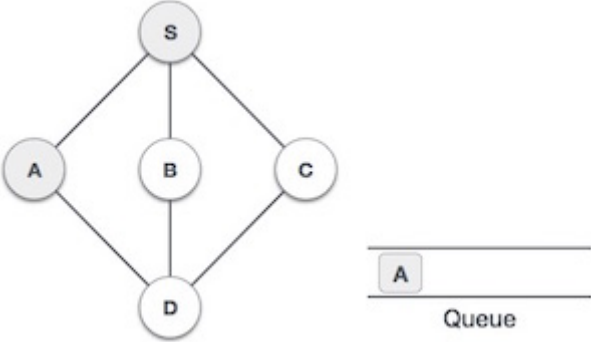
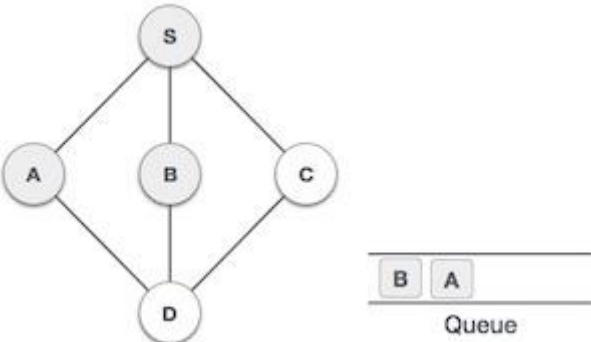
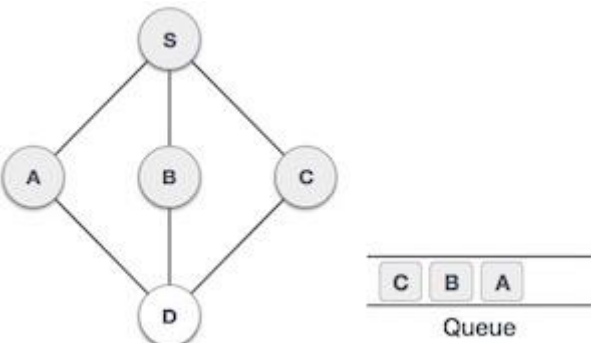
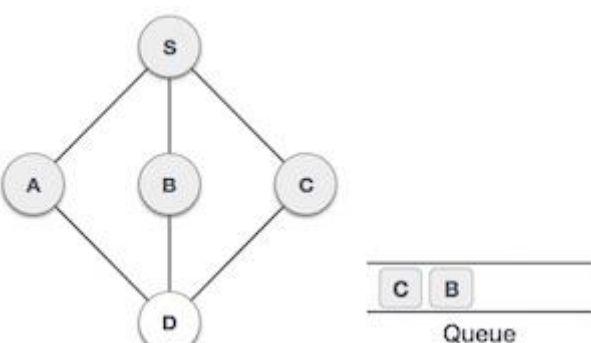
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

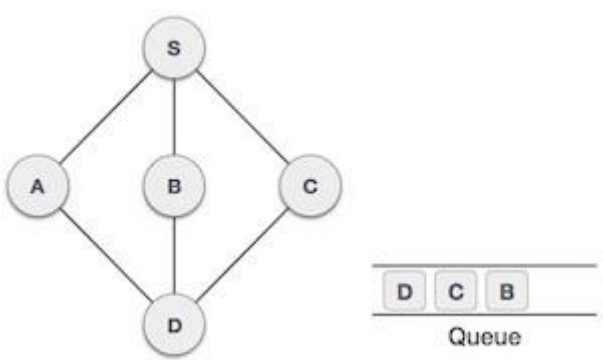


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.

Step	Traversal	Description
3		We then see an unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> , mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from <b>S</b> is <b>C</b> . We mark it as visited and enqueue it.
6		Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b> .

Step	Traversal	Description
7		From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over

Code:

```
# Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s. from collections import defaultdict
# This class represents a directed graph
# using adjacency list representation
class Graph:
    # Constructor
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)
        # function to add an edge to graph
        def addEdge(self,u,v):
            self.graph[u].append(v)
        # Function to print a BFS of graph
        def BFS(self, s):
            # Mark all the vertices as not visited
            visited = [False] * (max(self.graph) + 1)
```

```

# Create a queue for BFS
queue = []
# Mark the source node as
# visited and enqueue it
queue.append(s)
visited[s] = True
while queue:
    # Dequeue a vertex from
    # queue and print it
    s = queue.pop(0)
    print (s, end = " ")
    # Get all adjacent vertices of the
    # dequeued vertex s. If a adjacent
    # has not been visited, then mark it
    # visited and enqueue it
    for i in self.graph[s]:
        if visited[i] == False:
            queue.append(i)
            visited[i] = True

# Driver code
# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print ("Following is Breadth First Traversal"
      " (starting from vertex 2)")

```

g.BFS(2)

**OUTPUT:**

Following is Breadth First Traversal (starting from vertex 2)

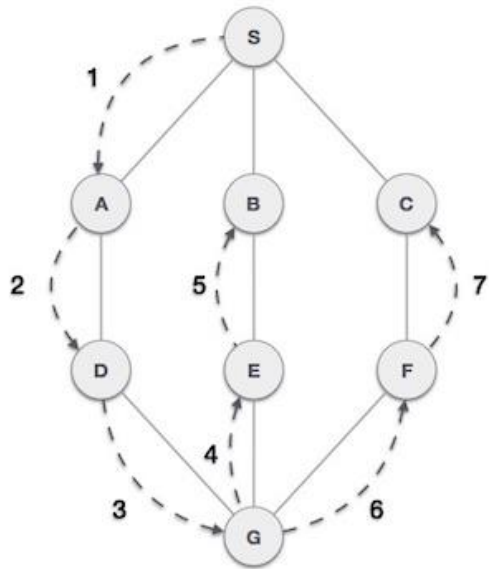
> 3

2 0 3 1 3

>

#### 4. Write a Program to Implement Depth First Search

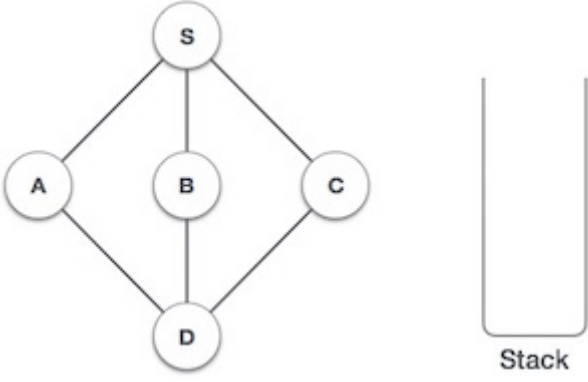
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

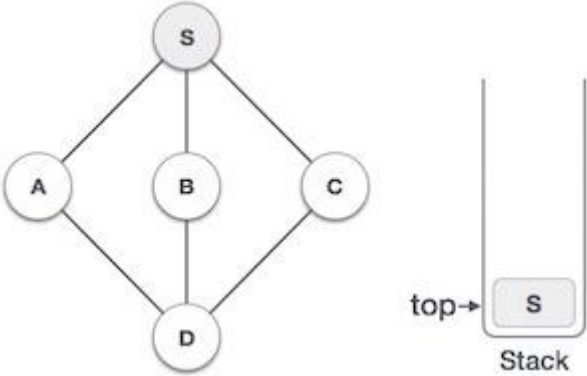
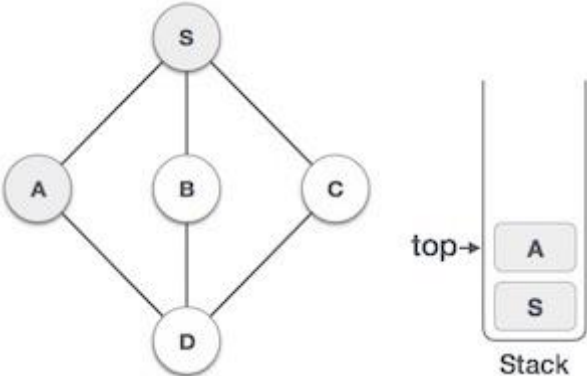
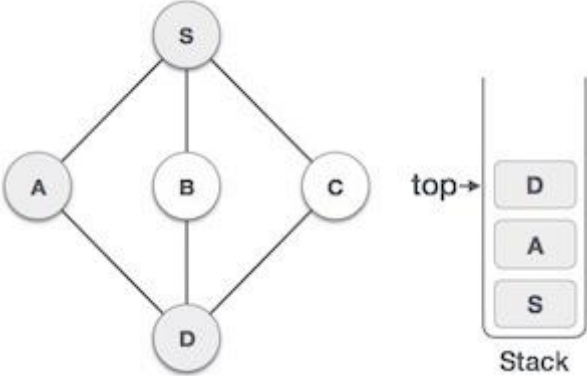


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

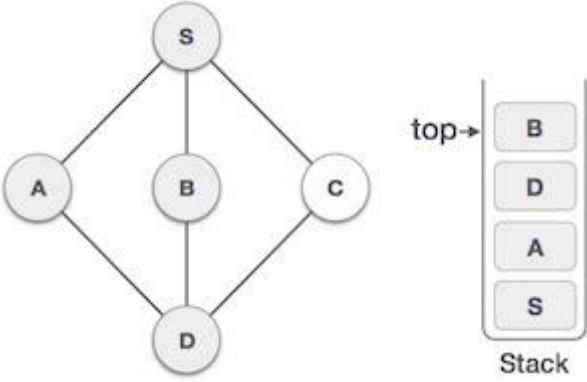
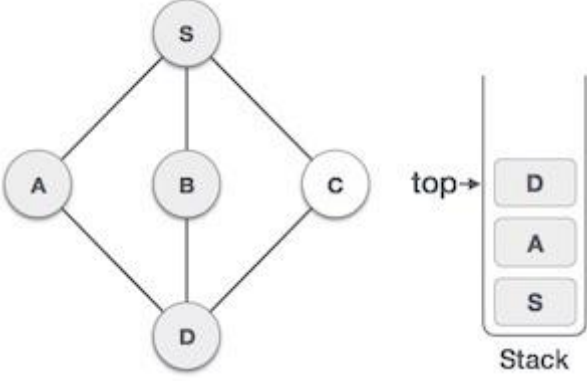
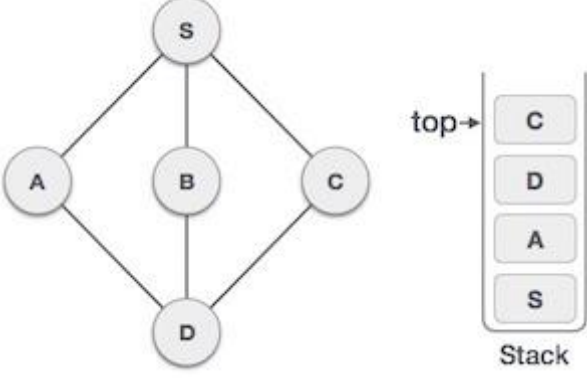
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the

stack is empty.

Step	Traversal	Description
1		Initialize the stack.

Step	Traversal	Description
2		Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>A</b> . Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.
4		Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.



Step	Traversal	Description
5		<p>We choose <b>B</b>, mark it as visited and put onto the stack.</p> <p>Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</b> from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it as visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

#### CODE:

```
# program to print DFS traversal from a given graph
from collections import defaultdict
```

```

# This class represents a directed graph using
# adjacency list representation
class Graph:
    # Constructor
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):
        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')
        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self, v):
        # Create a set to store visited vertices
        visited = set()
        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil(v, visited)

# Driver code
# Create a graph given

```

# in the above diagram

```
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

**OUTPUT:**

Following is Depth First Traversal (starting from vertex 2)

2 0 1 9 3

## 5. Write a Program to Implement 8-Puzzle problem

“It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state”.

The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle is a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it.

Initial State			Goal State		
1	2	3	2	8	1
8		4		4	3
7	6	5	7	6	5

A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration such as two situations given. The state of 8-puzzle is the different permutation of tiles within the frame. The operations are the permissible moves up, down, left, right. Here at each step of the problem a function  $f(x)$  will be defined which is the combination of  $g(x)$  and  $h(x)$ .

i.e.  $F(x) = g(x) + h(x)$

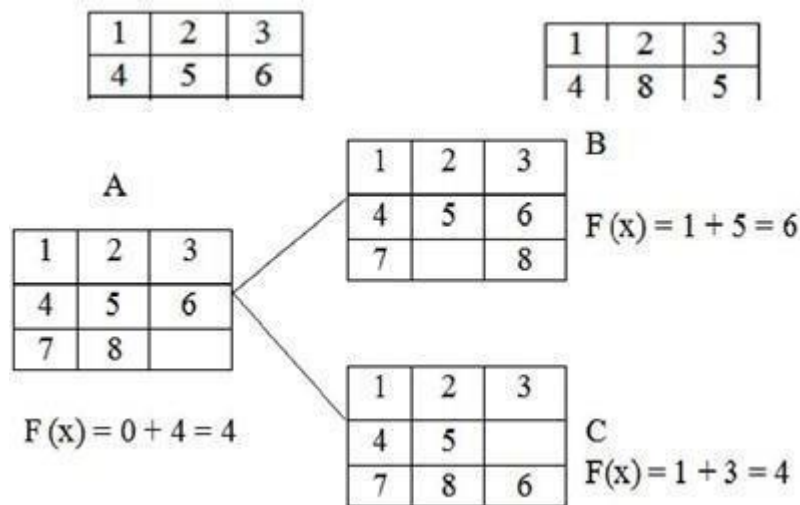
Where,  **$g(x)$** : how many steps in the problem you have already done or the current state from the initial state,  **$h(x)$** : Number of ways through which you can reach at the goal state from the current state

**$h(x)$** : is the heuristic estimator that compares the current state with the goal state note down how many states are displaced from the initial or the current state. After calculating the  **$f$**  value at each step finally take the smallest  **$f(x)$**  value at every step and choose that as the next current state to get the goal

Let us take an example.

Figure (Initial State)

(Goal State)



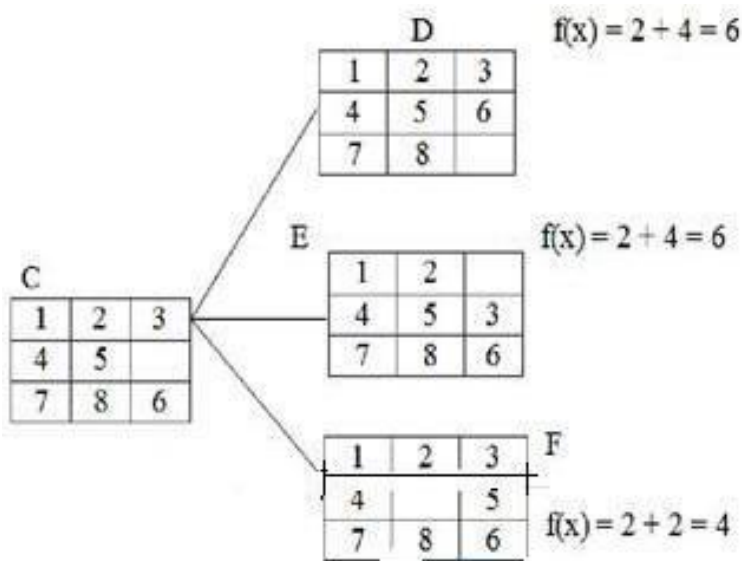
Step 1:

$f(x)$  is the step required to reach at the goal state from the initial state. So in the tray either 6 or 8 can change their portions to fill the empty position. So there will be two possible current states namely B and C. The  $f(x)$  value of B is 6 and that of C is 4. As 4 is the minimum, so take C as the current state to the next state.

Step 2:

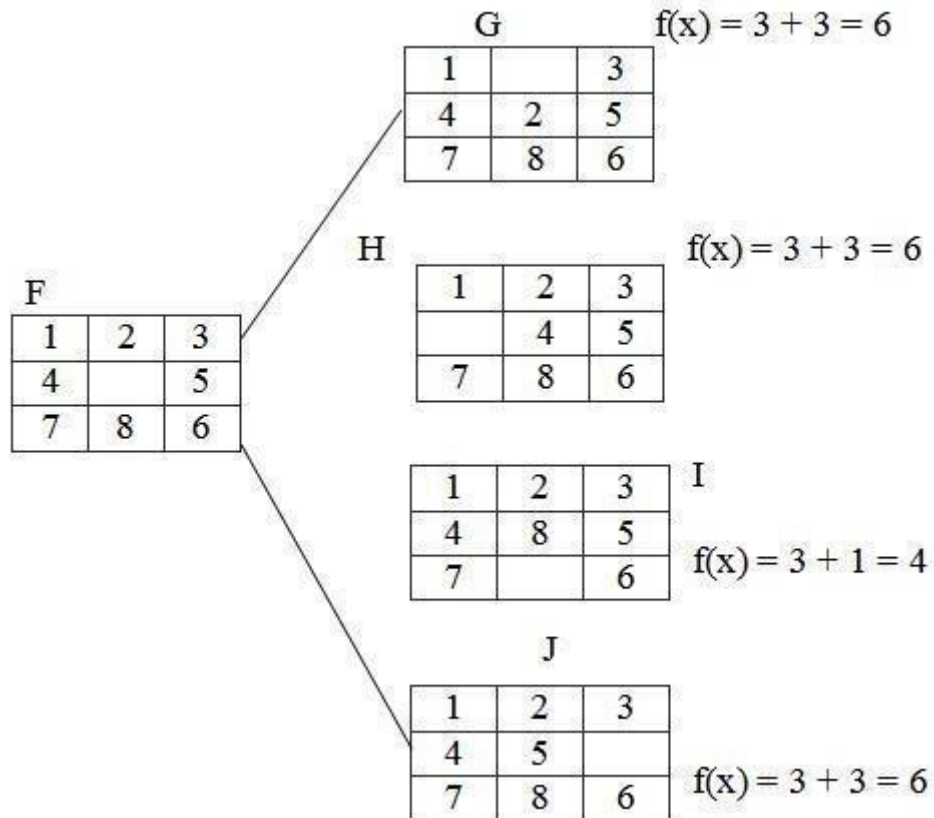
In this step, from the tray C three states can be drawn. The empty position will contain either 5 or 3 or 6. So for three different values three different states can be obtained. Then calculate each of their  $f(x)$  and take the minimum one.

Here the state F has the minimum value i.e. 4 and hence take that as the next current state.



Step 3:

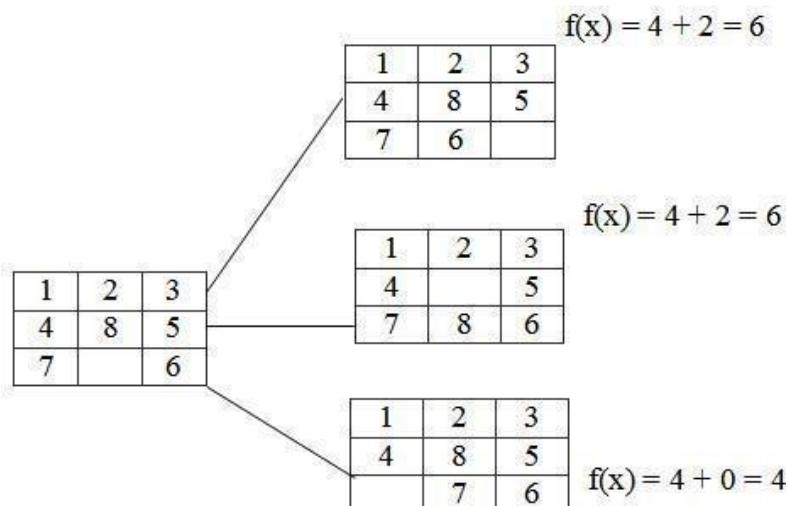
The tray F can have 4 different states as the empty positions can be filled with b4 values i.e.2, 4, 5, 8.



Step 4:

In the step-3 the tray I has the smallest  $f(n)$  value. The tray I can be implemented in 3 different states because the empty position can be filled by the members like 7, 8, 6.

Hence, we reached at the goal state after few changes of tiles in different positions of the trays.



This problem requires a lot of space for saving the different trays. Time complexity is more than that of other problems.

The user has to be very careful about the shifting of tiles in the trays. Very complex puzzle games can be solved by this technique.

#### CODE:

```
def solve(self, board):
    dict = {}
    flatten = []
    for i in range(len(board)):
        flatten += board[i]
    flatten = tuple(flatten)
    dict[flatten] = 0
    if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):
        return 0
    return self.get_paths(dict)

def get_paths(self, dict):
    cnt = 0
    while True:
        current_nodes = [x for x in dict if dict[x] == cnt]
        if len(current_nodes) == 0:
            return -1
        for node in current_nodes:
            next_moves = self.find_next(node)
            for move in next_moves:
                if move not in dict:
                    dict[move] = cnt + 1
                if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):
                    return cnt + 1
            cnt += 1

def find_next(self, node):
```

```

moves = {
0: [1, 3],
1: [0, 2, 4],
2: [1, 5],
3: [0, 4, 6],
4: [1, 3, 5, 7],
5: [2, 4, 8],
6: [3, 7],
7: [4, 6, 8],
8: [5, 7],
}
results = []
pos_0 = node.index(0)
for move in moves[pos_0]:
    new_node = list(node)
    new_node[move], new_node[pos_0] = new_node[pos_0], new_node[move]
    results.append(tuple(new_node))
return results
ob = Solution()
matrix = [
    [3, 1, 2],
    [4, 7, 5],
    [6, 8, 0]
]
print(ob.solve(matrix))

```

**INPUT:**

```

matrix = [
    [3, 1, 2],
    [4, 7, 5],

```



[6, 8, 0] ]

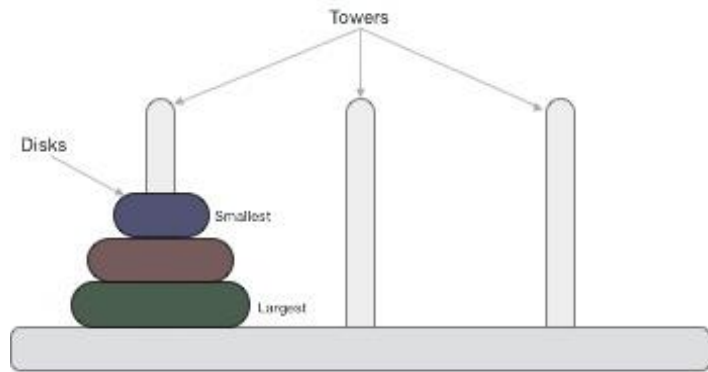
**OUTPUT:**

4

## 6. Implementation of Towers of Hanoi Problem

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –

These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

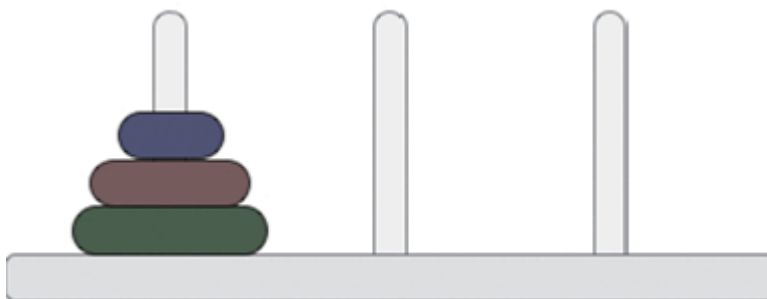


### Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.

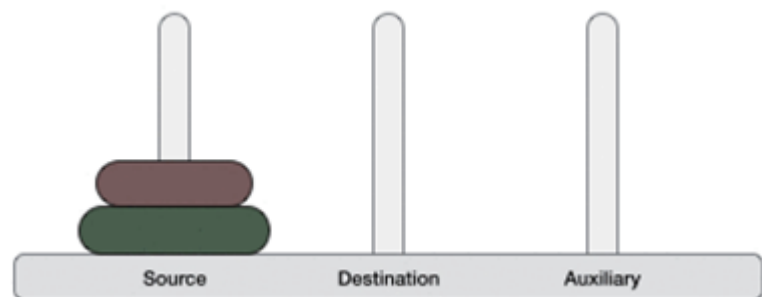


Tower of Hanoi puzzle with  $n$  disks can be solved in minimum  $2^n - 1$  steps. This presentation shows that a puzzle with 3 disks has taken  $2^3 - 1 = 7$  steps.

First its needed to learn how to solve this problem with lesser amount of disks, say  $\rightarrow$  1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If there is only one disk, then it can easily be moved from source to destination peg.

If there are 2 disks –

- First, move the smaller (top) disk to aux peg.
- Then, move the larger (bottom) disk to destination peg.
- And finally, move the smaller disk from aux to destination peg.



We divide the stack of disks in two parts. The largest disk ( $n^{\text{th}}$  disk) is in one part and all other ( $n-1$ ) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other ( $n-1$ ) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

**Step 1** – Move  $n-1$  disks from **source** to **aux**

**Step 2** – Move  $n^{\text{th}}$  disk from **source** to **dest**

**Step 3** – Move  $n-1$  disks from **aux** to **dest**

**Code:**

```
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print("Move disk 1 from rod",from_rod,"to rod",to_rod)
        return
```

```

TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)

print("Move disk",n,"from rod",from_rod,"to rod",to_rod)

TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

```

### # Driver code

```
n = 4
```

```
TowerOfHanoi(n, 'A', 'C', 'B')
```

# A, C, B are the name of rods

### Output:

Tower of Hanoi Solution for 4 disks:

A: [4, 3, 2, 1] B: [] C: []

Move disk1 from rod A to rod B

A: [4, 3, 2] B: [1] C: []

Move disk2 from rod A to rod C

A: [4, 3] B: [1] C: [2]

Move disk1 from rod B to rod C

A: [4, 3] B: [] C: [2, 1]

Move disk3 from rod A to rod B

A: [4] B: [3] C: [2, 1]

Move disk1 from rod C to rod A

A: [4, 1] B: [3] C: [2]

Move disk2 from rod C to rod B

A: [4, 1] B: [3, 2] C: []

Move disk1 from rod A to rod B

A: [4] B: [3, 2, 1] C: []

Move disk4 from rod A to rod C

A: [] B: [3, 2, 1] C: [4]

Move disk1 from rod B to rod C

A: [] B: [3, 2] C: [4, 1]

Move disk2 from rod B to rod A

A: [2] B: [3] C: [4, 1]

Move disk1 from rod C to rod A

A: [2, 1] B: [3] C: [4]

Move disk3 from rod B to rod C

A: [2, 1] B: [] C: [4, 3]

Move disk1 from rod A to rod B

A: [2] B: [1] C: [4, 3]

Move disk2 from rod A to rod C

A: [] B: [1] C: [4, 3, 2]

Move disk1 from rod B to rod C

A: [] B: [] C: [4, 3, 2, 1]

## 7. Write a Program to Implement Missionaries-Cannibals Problem

Missionaries and Cannibals problem is very famous in Artificial Intelligence because it was the subject of the first paper that approached problem formulation from an analytical viewpoint. The problem can be stated as follow.

Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Now we have to find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in another side. The above problem can be solved by a graph search method. Here I represent the problem as a set of states and operators. States are snapshots of the world and operators are those which transform one state into another state. States can be mapped to nodes of a graph and operators are the edges of the graph.

### Explanation

Missionaries and Cannibals can be solved by using different search algorithms like Breadth first and Depth first search algorithm to find the solution. The node of the graph to be searched is represented by a state space. Each state space can be represent by

State(no\_of\_missionaries, no\_of\_cannibals, side\_of\_the\_boat)

Where no\_of\_missionaries are the number of missionaries at left side of river, no\_of\_cannibals are the number of cannibals at the left side of river and side\_of\_the\_boat is the side of the boat at particular state. For our case

Initial State => State(3, 3, 0) and

Final State => State(0, 0, 1).

Where 0 represents left side and 1 represents right side of river. We should make a graph search which traverse the graph from initial state and find out the final state in fewest moves. There are many AI searches that search the graphs like Breadth first search, Depth first search, or iterative deepening search. Each of these different search methods has different properties such as whether a result is guaranteed, and how much time and space is needed to carry out the search. This project uses Breadth first and Depth first search.

### Possible Moves

A move is characterized by the number of missionaries and the number of cannibals taken in the boat at one time. Since the boat can carry no more than two people at once, the only feasible combinations are:

Carry (2, 0).

Carry (1, 0).

Carry (1, 1).

Carry (0, 1).

Carry(0, 2).

Where Carry (M, C) means the boat will carry M missionaries and C cannibals on one trip.

### Feasible Moves

Once we have found a possible move, we have to confirm that it is feasible. It is not a feasible to move more missionaries or more cannibals than that are present on one bank.

- When the state is state(M1, C1, left) and we try carry (M,C) then  $M \leq M1$  and  $C \leq C1$  must be true.
- When the state is state(M1, C1, right) and we try carry(M, C) then  $M + M1 \leq 3$  and  $C + C1 \leq 3$  must be true.

### Legal Moves

Once we have found a feasible move, we must check that is legal i.e. no missionaries must be eaten.

Legal(X, X).

Legal(3, X).

Legal(0, X).

The only safe combinations are when there are equal numbers of missionaries and cannibals or all the missionaries are on one side. Generating the next state Above figure only shows valid states.

**CODE:**

```

''' mclib.py '''
class MCState:
    ### MC is missionaries and cannibals
    def __init__(self, state_vars, num_moves=0, parent=None):
        self.state_vars = state_vars
        self.num_moves = num_moves
        self.parent = parent
    ### decorator
    @classmethod
    def root(cls):
        return cls((3,3,1))
    def get_possible_moves(self):
        ''' return all possible moves in the game as tuples
        possible moves:
        1 or 2 mis
        1 or 2 cannibals
        1 mis, 1 can
        '''
        moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
        return moves
    def is_legal(self):
        missionaries = self.state_vars[0]
        cannibals = self.state_vars[1]
        ## could have done tuple unpacking too:
        ## missionaries, cannibals, boat = self.state_vars
        if missionaries < 0 or missionaries > 3:
            return False
        elif cannibals < 0 or cannibals > 3:
            return False
        return True

```



```

## alternate
# if 0 <= missionaries <= 3 and 0 <= cannibals <= 3
# return True
###
def is_solution(self):
    if self.state_vars == (0,0,0):
        return True
    return False

def is_failure(self):
    missionaries = self.state_vars[0]
    cannibals = self.state_vars[1]
    boat = self.state_vars[2]
    ## could have done tuple unpacking too:
    ## missionaries, cannibals, boat = self.state_vars

    ### missionaries on right side AND more cannibals than missionaries
    if missionaries > 0 and missionaries < cannibals:
        return True

    ## to make this easier to understand, I will create temporary variables
    ## but we could just substitute the math and skip the variables
    missionaries_on_left = 3 - missionaries
    cannibals_on_left = 3 - cannibals
    if missionaries_on_left > 0 and missionaries_on_left < cannibals_on_left:
        return True

    ## if you replace the math in, you get:
    #if 3 - missionaries > 0 and 3 - missionaries < 3 - cannibals
    # which leads to:
    #if missionaries < 3 and cannibals < missionaries:

```

```

    ### if we make it here, we aren't in a failed state!
    return False

def get_next_states(self):
    ## using possible move, get next states
    moves = self.get_possible_moves()
    all_states = list()
    mis_right, can_right, raft_right = self.state_vars
    ## if raft is on right, subtract move from these numbers
    ## if raft is on left, add these move numbers to these numbers
    for move in moves:
        change_mis, change_can = move
        if raft_right == 1: ## mis_right = 3; can_right = 3, raft_right = 1
            new_state_vars = (mis_right-change_mis, can_right-change_can, 0)
        else:
            new_state_vars = (mis_right+change_mis, can_right+change_can, 1)

        ## notice the number of moves is increasing by 1
        ## also notice we are passing self to our child.
        new_state = MCState(new_state_vars, self.num_moves+1, self)
        if new_state.is_legal():
            all_states.append(new_state)
    return all_states

def __str__(self):
    return "MCState[{}]" .format(self.state_vars)

def __repr__(self):
    return str(self)

def search(dfs=True):
    ### this is the stack/queue that we used before from collections import deque
    ### create the root state
    root = MCState.root()

```

```

### we use the stack/queue for keeping track of where to search next
to_search = deque()

### use a set to keep track of where we've been
seen_states = set()

### use a list to keep track of the solutions that have been seen
solutions = list()

### start the search with the root
to_search.append(root)

### safety variable for infinite loops!
loop_count = 0
max_loop = 10000

### while the stack/queue still has items
while len(to_search) > 0:
    loop_count += 1
    if loop_count > max_loop:
        print(len(to_search))
        print("Escaping this super long loop!")
        break

    ### get the next item
    current_state = to_search.pop()

    ## look at the current state's children

    ## this uses the rule for actions and moves to create next states
    ## it is also removing all illegal states
    next_states = current_state.get_next_states()

    ## next_states is a list, so iterate through it
    for possible_next_state in next_states[::-1]:

        ## to see if we've been here before, we look at the state variables
        possible_state_vars = possible_next_state.state_vars

```

```

## we use the set and the "not in" boolean comparison
if possible_state_vars not in seen_states:
    if possible_next_state.is_failure():
        #print("Failure!")
        continue
    elif possible_next_state.is_solution():
        ## Save it into our solutions list
        solutions.append(possible_next_state)
        #print("Solution!")
        continue
    # the state variables haven't been seen yet
    # so we add the state itself into the searching stack/queue

#### IMPORTANT
## which side we append on changes how the search works
## why is this?
if dfs:
    to_search.append(possible_next_state)
else:
    to_search.appendleft(possible_next_state)
# now that we have "seen" the state, we add the state vars to the set.
# this means next time when we do the "not in", that will return False
# because it IS in
#seen_states.add(possible_state_vars)
seen_states.add(possible_state_vars)

## finally, we reach this line when the stack/queue is empty
## (len (to_searching==))
print("Found {} solutions".format(len(solutions)))
return solutions
sol_dfs = search(True)

```

```

sol_bfs = search(False)
current_state = sol_dfs[0]
while current_state:
    print(current_state)
    current_state = current_state.parent

```

```

print("--")
current_state = sol_dfs[1]
while current_state:
    print(current_state)
    current_state = current_state.parent

```

```

print("--")
current_state = sol_bfs[0]
while current_state:
    print(current_state)
    current_state = current_state.parent

```

```

print("--")
current_state = sol_bfs[1]
while current_state:
    print(current_state)
    current_state = current_state.parent

```

Found 2 solutions

Found 2 solutions

MCState[(0, 0, 0)]

MCState[(1, 1, 1)]

MCState[(0, 1, 0)]

MCState[(0, 3, 1)]

MCState[(0, 2, 0)]

MCState[(2, 2, 1)]

MCState[(1, 1, 0)]

MCState[(3, 1, 1)]

MCState[(3, 0, 0)]

MCState[(3, 2, 1)]

MCState[(3, 1, 0)]

MCState[(3, 3, 1)]

--

MCState[(0, 0, 0)]

MCState[(0, 2, 1)]

MCState[(0, 1, 0)]

MCState[(0, 3, 1)]

MCState[(0, 2, 0)]

MCState[(2, 2, 1)]

MCState[(1, 1, 0)]

MCState[(3, 1, 1)]

MCState[(3, 0, 0)]

MCState[(3, 2, 1)]

MCState[(3, 1, 0)]

MCState[(3, 3, 1)]

--

MCState[(0, 0, 0)]

MCState[(0, 2, 1)]

MCState[(0, 1, 0)]

MCState[(0, 3, 1)]

MCState[(0, 2, 0)]

MCState[(2, 2, 1)]

MCState[(1, 1, 0)]

MCState[(3, 1, 1)]

MCState[(3, 0, 0)]

MCState[(3, 2, 1)]

MCState[(2, 2, 0)]

MCState[(3, 3, 1)]

--

MCState[(0, 0, 0)]

MCState[(1, 1, 1)]

MCState[(0, 1, 0)]

MCState[(0, 3, 1)]

MCState[(0, 2, 0)]

MCState[(2, 2, 1)]

MCState[(1, 1, 0)]

MCState[(3, 1, 1)]

MCState[(3, 0, 0)]

MCState[(3, 2, 1)]

MCState[(2, 2, 0)]

MCState[(3, 3, 1)]

OUTPUT: Disks are set

## 8. Write a Program to Implement Travelling Salesman Problem

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For  $n$  number of vertices in a graph, there are  $(n - 1)!$  number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph  $G = (V, E)$ , where  $V$  is a set of cities and  $E$  is a set of weighted edges. An edge  $e(u, v)$  represents that vertices  $u$  and  $v$  are connected. Distance between vertex  $u$  and  $v$  is  $d(u, v)$ , which should be non-negative.

Suppose we have started at city 1 and after visiting some cities now we are in city  $j$ . Hence, this is a partial tour. We certainly need to know  $j$ , since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities  $S \in \{1, 2, 3, \dots, n\}$  that includes 1, and  $j \in S$ , let  $C(S, j)$  be the length of the shortest path visiting each node in  $S$  exactly once, starting at 1 and ending at  $j$ .

When  $|S| > 1$ , we define  $C(S, 1) = \infty$  since the path cannot start and end at 1.

Now, let express  $C(S, j)$  in terms of smaller sub-problems. We need to start at 1 and end at  $j$ . We should select the next city in such a way that

$$C(S, j) = \min_{i \in S, i \neq j} \{C(S - \{j\}, i) + d(i, j)\}$$

### Algorithm: Traveling-Salesman-Problem

$$C(\{1\}, 1) = 0$$

for  $s = 2$  to  $n$  do

    for all subsets  $S \in \{1, 2, 3, \dots, n\}$  of size  $s$  and containing 1

$$C(S, 1) = \infty$$

    for all  $j \in S$  and  $j \neq 1$



$$C(S, j) = \min\{C(S - \{j\}, i) + d(i, j) \text{ for } i \in S \text{ and } i \neq j\}$$

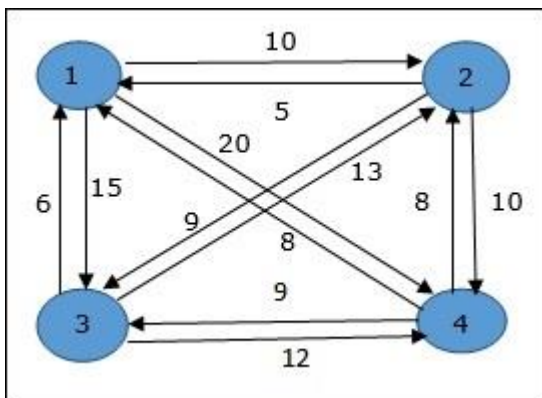
Return  $\min_j C(\{1, 2, 3, \dots, n\}, j) + d(j, i)$

### Analysis

There are at the most  $2^n \cdot n$  sub-problems and each one takes linear time to solve. Therefore, the total running time is  $O(2^n \cdot n^2)$

### Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$S = \Phi$

$$\text{Cost}(2, \Phi, 1) = d(2, 1) = 5 \quad \text{Cost}(2, \Phi, 1) = d(2, 1) = 5 \quad \text{Cost}(2, \Phi, 1) = d(2, 1) = 5 \quad \text{Cost}(2, \Phi, 1) = d(2, 1) = 5$$

$$\text{Cost}(3, \Phi, 1) = d(3, 1) = 6 \quad \text{Cost}(3, \Phi, 1) = d(3, 1) = 6 \quad \text{Cost}(3, \Phi, 1) = d(3, 1) = 6 \quad \text{Cost}(3, \Phi, 1) = d(3, 1) = 6$$

$$\text{Cost}(4, \Phi, 1) = d(4, 1) = 8 \quad \text{Cost}(4, \Phi, 1) = d(4, 1) = 8 \quad \text{Cost}(4, \Phi, 1) = d(4, 1) = 8 \quad \text{Cost}(4, \Phi, 1) = d(4, 1) = 8$$

$S = 1$

$$\text{Cost}(i, s) = \min\{\text{Cost}(j, s - \{j\}) + d[i, j]\} \quad \text{Cost}(i, s) = \min\{\text{Cost}(j, s - \{j\}) + d[i, j]\}$$

$$\text{Cost}(2, \{3\}, 1) = d[2, 3] + \text{Cost}(3, \Phi, 1) = 9 + 6 = 15 \quad \text{cost}(2, \{3\}, 1) = d[2, 3] + \text{cost}(3, \Phi, 1) = 9 + 6 = 15$$

$$\text{Cost}(2, \{4\}, 1) = d[2, 4] + \text{Cost}(4, \Phi, 1) = 10 + 8 = 18 \quad \text{cost}(2, \{4\}, 1) = d[2, 4] + \text{cost}(4, \Phi, 1) = 10 + 8 = 18$$

$$\text{Cost}(3, \{2\}, 1) = d[3, 2] + \text{Cost}(2, \Phi, 1) = 13 + 5 = 18 \quad \text{cost}(3, \{2\}, 1) = d[3, 2] + \text{cost}(2, \Phi, 1) = 13 + 5 = 18$$

$$\text{Cost}(3, \{4\}, 1) = d[3, 4] + \text{Cost}(4, \Phi, 1) = 12 + 8 = 20 \quad \text{cost}(3, \{4\}, 1) = d[3, 4] + \text{cost}(4, \Phi, 1) = 12 + 8 = 20$$

$$\text{Cost}(4, \{3\}, 1) = d[4, 3] + \text{Cost}(3, \Phi, 1) = 9 + 6 = 15 \quad \text{cost}(4, \{3\}, 1) = d[4, 3] + \text{cost}(3, \Phi, 1) = 9 + 6 = 15$$

$$\text{Cost}(4, \{2\}, 1) = d[4, 2] + \text{Cost}(2, \Phi, 1) = 8 + 5 = 13 \quad \text{cost}(4, \{2\}, 1) = d[4, 2] + \text{cost}(2, \Phi, 1) = 8 + 5 = 13$$

**S = 2**

$$\text{Cost}(2, \{3, 4\}, 1) =$$

$$d[2, 3] + \text{Cost}(3, \{4\}, 1) = 9 + 20 = 29$$

$$d[2, 4] + \text{Cost}(4, \{3\}, 1) = 10 + 15 = 25 = 25 \quad \text{Cost}(2, \{3, 4\}, 1)$$

$$\{d[2, 3] + \text{cost}(3, \{4\}, 1) = 9 + 20 = 29 \quad d[2, 4] + \text{Cost}(4, \{3\}, 1) = 10 + 15 = 25$$

$$= 25$$

$$\text{Cost}(3, \{2, 4\}, 1) =$$

$$d[3, 2] + \text{Cost}(2, \{4\}, 1) = 13 + 18 = 31$$

$$d[3, 4] + \text{Cost}(4, \{2\}, 1) = 12 + 13 = 25 = 25 \quad \text{Cost}(3, \{2, 4\}, 1)$$

$$\{d[3, 2] + \text{cost}(2, \{4\}, 1) = 13 + 18 = 31$$

$$d[3, 4] + \text{Cost}(4, \{2\}, 1) = 12 + 13 = 25$$

$$= 25$$

$$\text{Cost}(4, \{2, 3\}, 1) =$$

$$d[4, 2] + \text{Cost}(2, \{3\}, 1) = 8 + 15 = 23$$

$$d[4, 3] + \text{Cost}(3, \{2\}, 1) = 9 + 18 = 27 = 23 \quad \text{Cost}(4, \{2, 3\}, 1)$$

$$\{d[4, 2] + \text{cost}(2, \{3\}, 1) = 8 + 15 = 23$$

$$d[4, 3] + \text{Cost}(3, \{2\}, 1) = 9 + 18 = 27$$

$$= 23$$

**S = 3**

$$\text{Cost}(1, \{2, 3, 4\}, 1) =$$

$$d[1, 2] + \text{Cost}(2, \{3, 4\}, 1) = 10 + 25 = 35$$

$$d[1, 3] + \text{Cost}(3, \{2, 4\}, 1) = 15 + 25 = 40$$

$$d[1, 4] + \text{Cost}(4, \{2, 3\}, 1) = 20 + 23 = 43 = 35 \quad \text{cost}(1, \{2, 3, 4\}, 1)$$

$$d[1, 2] + \text{cost}(2, \{3, 4\}, 1) = 10 + 25 = 35$$

$$d[1, 3] + \text{cost}(3, \{2, 4\}, 1) = 15 + 25 = 40$$

$$d[1,4] + \text{cost}(4, \{2,3\}, 1) = 20 + 23 = 43$$

$$= 35$$

The minimum cost path is 35.

Start from cost  $\{1, \{2, 3, 4\}, 1\}$ , we get the minimum value for  $d[1, 2]$ . When  $s = 3$ , select the path from 1 to 2 (cost is 10) then go backwards. When  $s = 2$ , we get the minimum value for  $d[4, 2]$ . Select the path from 2 to 4 (cost is 10) then go backwards.

When  $s = 1$ , we get the minimum value for  $d[4, 3]$ . Selecting path 4 to 3 (cost is 9), then we shall go to then go to  $s = \Phi$  step. We get the min. value for  $d[3, 1]$  (cost is 6).



#### CODE:

```

# program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    # store minimum weight
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        # store current Path weight(cost)
        current_pathweight = 0

```

```

# compute current path weight
k = s
for j in i:
    current_pathweight += graph[k][j]
    k = j
current_pathweight += graph[k][s]
# update minimum
min_path = min(min_path, current_pathweight)
return min_path

# Driver Code
if __name__ == "__main__":
    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))

```

**OUTPUT:**

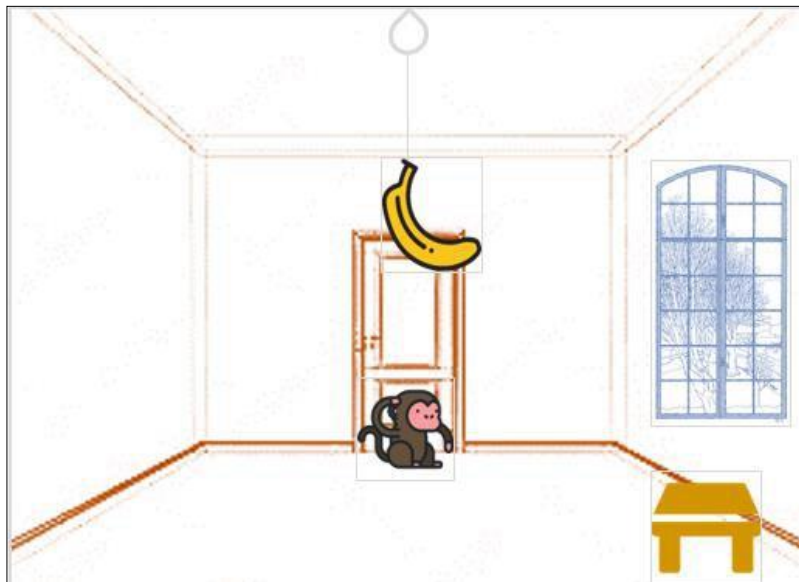
80

## 9. Write a Program to Implement Monkey-Banana Problem

Suppose the problem is as given below –

- A hungry monkey is in a room, and he is near the door.
- The monkey is on the floor.
- Bananas have been hung from the center of the ceiling of the room.
- There is a block (or chair) present in the room near the window.
- The monkey wants the banana, but cannot reach it.

So how can the monkey get the bananas?



So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case –

- Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both

the monkey and the block are on the floor.

- If the block position is not at the center, then monkey can drag it to the center.
- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

Now, let us see how we can solve this using Prolog. We will create some predicates as follows –

We have some predicates that will move from one state to another state, by performing action.

- When the block is at the middle, and monkey is on top of the block, and monkey does not have the banana (i.e. *has not* state), then using the *grasp* action, it will change from *has not* state to *have* state.
- From the floor, it can move to the top of the block (i.e. *on top* state), by performing the action *climb*.
- The **push** or **drag** operation moves the block from one place to another.
- Monkey can move from one place to another using **walk** or **move** clauses.

Another predicate will be canget(). Here we pass a state, so this will perform move predicate from one state to another using different actions, then perform canget() on state 2. When we have reached to the state '**has>**', this indicates '**has banana**'. We will stop the execution.

#### CODE:

```
def solve(banana,box,height,monkey,hold):
    if monkey==banana and height==1:
        ans.append("Monkey took banana")
        return True
    if (banana,box,height,monkey,hold) in d:
        return False
    found=0
    d[(banana,box,height,monkey,hold)]=1
    options={1:"Move to box", 2:"Move to banana", 3:"Climb onto the box",
4:"Hold box to move"}
    for option in options:
        if option==1 and hold==0 and height==0 and
solve(banana,box,height,box,hold):
        ans.append(options[option])
```

```

        found=1
        break
    elif option==2 and height==0 and ((hold==1 and
solve(banana,banana,height,banana,hold)) or (hold==0 and
solve(banana,box,height,banana,hold))):
        ans.append(options[option])
        found=1
        break
    elif option==3 and height==0 and monkey==box and
solve(banana,box,height+1,monkey,0):
        ans.append(options[option])
        found=1
        break
    elif option==4 and height==0 and monkey==box and
solve(banana,box,height,monkey,1):
        ans.append(options[option])
        found=1
        break
    return found

```

```

n=int(input("Enter the size of the world: "))
world=[[0]*n for i in range(n)]
x,y=map(int,input("Enter tree position: ").split())
world[x][y]=1
x,y=map(int,input("Enter box position: ").split())
world[x][y]=-1
x,y=map(int,input("Enter monkey position: ").split())
for i in range(n):
    for j in range(n):
        if world[i][j]==1:
            print(f"Monkey found the banana tree at ({i},{j})")

```

```
        banana=(i,j)
    if world[i][j]==-1:
        print(f"Box found at ({i},{j})")
        box=(i,j)
    d={}
    ans=[]
    solve(banana,box,0,(x,y),0)
    ans.reverse()
    print(*ans,sep="\n")
```

**OUTPUT:**

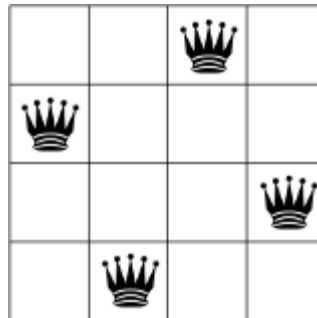
```
5
2 2
4 4
0 0
```



## 10. Write a Program to Implement N-Queens Problem

In the following sections, we'll illustrate constraint programming (CP) by a combinatorial problem based on the game of chess. In chess, a queen can attack horizontally, vertically, and diagonally. The N-queens problem is how can N queens be placed on an  $N \times N$  chessboard so that no two of them attack each other?

Below, you can see one possible solution to the N-queens problem for  $N = 4$ .



No two queens are on the same row, column, or diagonal.

Note that this isn't an optimization problem: we want to find all possible solutions, rather than one optimal solution, which makes it a natural candidate for constraint programming. The following sections describe the CP approach to the N-queens problem, and present Python programs that solve it using both the CP-SAT solver and the original CP solver.

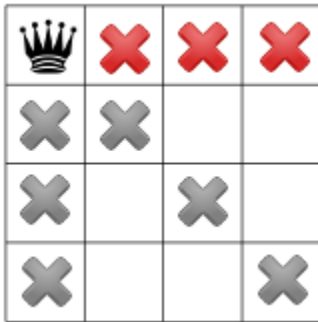
### CP approach to the N-queens problem

A CP solver works by systematically trying all possible assignments of values to the variables in a problem, to find the feasible solutions. In the 4-queens problem, the solver starts at the leftmost column and successively places one queen in each column, at a location that is not attacked by any previously placed queens.

Propagation and backtracking

There are two key elements to a constraint programming search:

**Propagation** – Each time the solver assigns a value to a variable, the constraints add restrictions on the possible values of the unassigned variables. These restrictions *propagate* to future variable assignments. For example, in the 4-queens



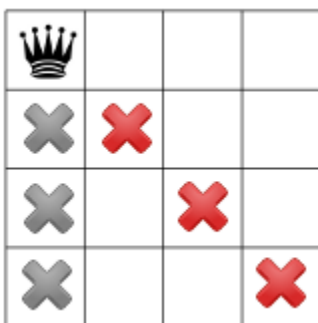
problem, each time the solver places a queen, it can't place any other queens on the row and diagonals the current queen is on. Propagation can speed up the search significantly by reducing the set of variable values the solver must explore.

**Backtracking** occurs when either the solver can't assign a value to the next variable, due to the constraints, or it finds a solution. In either case, the solver backtracks to a previous stage and changes the value of the variable at that stage to a value that hasn't already been tried. In the 4-queens example, this means moving a queen to a new square on the current column.

Next, you'll see how constraint programming uses propagation and backtracking to solve the 4-queens problem.

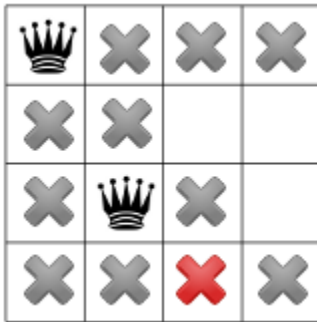
Let's suppose the solver starts by arbitrarily placing a queen in the upper left corner. That's a hypothesis of sorts; perhaps it will turn out that no solution exists with a queen in the upper left corner.

Given this hypothesis, what constraints can we propagate? One constraint is that there can be only one queen in a column (the gray Xs below), and another constraint prohibits two queens on the same diagonal (the red Xs below).

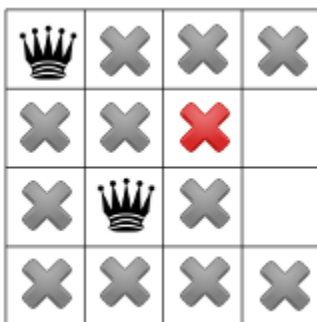


Our third constraint prohibits queens on the same row:

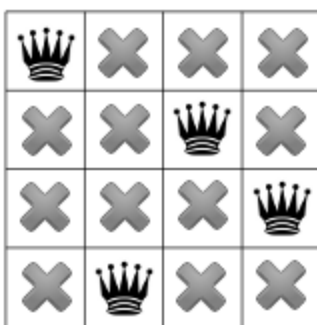
Our constraints propagated, we can test out another hypothesis, and place a second queen on one of the available remaining squares. Our solver might decide to place in it the first available square in the second column:



After propagating the diagonal constraint, we can see that it leaves no available squares in either the third column or last row:



With no solutions possible at this stage, we need to backtrack. One option is for the solver to choose the other available square in the second column. However, constraint propagation then forces a queen into the second row of the third column, leaving no valid spots for the fourth queen:



And so the solver must backtrack again, this time all the way back to the placement of the first queen. We have now shown that no solution to the queens problem will occupy a corner square.

Since there can be no queen in the corner, the solver moves the first queen down by one, and propagates, leaving only one spot for the second queen:

×	×		
♔	×	×	×
×	×		
×	♔	×	

Propagating again reveals only one spot left for the third queen:

×	×	♔	
♔	×	×	×
×	×	×	
×	♔	×	

And for the fourth and final queen:

×	×	♔	×
♔	×	×	×
×	×	×	♔
×	♔	×	×

We have our first solution! If we instructed our solver to stop after finding the first solution, it would end here. Otherwise, it would backtrack again and place the first queen in the third row of the first column.

#### CODE:

# Problem using backtracking

global N

N = 4

```
def printSolution(board):

    for i in range(N):

        for j in range(N):

            print board[i][j],

        print

# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens

def isSafe(board, row, col):

    # Check this row on left side

    for i in range(col):

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

            return False

    # Check upper diagonal on left side

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

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

            return False

    # Check lower diagonal on left side
```

```

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

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

        return False

return True

def solveNQUtil(board, col):

    # base case: If all queens are placed

    # then return true

    if col >= N:

        return True

    # Consider this column and try placing

    # this queen in all rows one by one

    for i in range(N):

        if isSafe(board, i, col):

            # Place this queen in board[i][col]

            board[i][col] = 1

            # recur to place rest of the queens

            if solveNQUtil(board, col + 1) == True:

                return True

            # If placing queen in board[i][col]

            # doesn't lead to a solution, then

            # queen from board[i][col]

            board[i][col] = 0

```

```

    # if the queen can not be placed in any row in
    # this column col then return false

    return False

# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.

def solveNQ():
    board = [ [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0]
             ]

    if solveNQUtil(board, 0) == False:
        print "Solution does not exist"

        return False

    printSolution(board)

```

```
        return True

# driver program to test above function
solveNQ()
```

**OUTPUT:**

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0