```python
#DFS
graph={
    'A': ['B','C'],
    'B':['D','E'],
    'C':['F','G'],
    'D':[],
    'E':[],
    'F':[],
    'G':[],
}
visited=set()
def dfs(visited, graph, node):
  if node not in visited:
    print(node)
    visited.add(node)
    for neighbour in graph[node]:
      dfs(visited, graph, neighbour)
print("Following is the DFS")
dfs(visited, graph, 'A')
```

```
Following is the DFS
A
B
D
E
C
F
G
```

```
#8 puzzle using BFS
import numpy as np
import pandas as pd
import os
def bfs(src, target):
  queue=[]
  queue.append(src)
  exp=[]
  while len(queue)>0:
    source=queue.pop(0)
    exp.append(source)
    print(source)
    if source==target:
      print("success")
      return
    poss_moves=[]
    poss_moves=possible_moves(source, exp)
    for move in poss_moves:
      if move not in exp and move not in queue:
        queue.append(move)

def possible_moves(state, visited_states):
  b=state.index(0)
  d=[]
  if b not in [0,1,2]:
    d.append('u')
  if b not in [6,7,8]:
    d.append('d')
  if b not in [0,3,6]:
    d.append('l')
  if b not in [2,5,8]:
    d.append('r')
  pos_moves=[]
  for i in d:
    pos_moves.append(gen(state,i,b))

  return [move_it_can for move_it_can in pos_moves if move_it_can not in visited_states]

def gen(state, m, b):
  temp=state.copy()
  if m=='d':
    temp[b+3], temp[b]=temp[b],temp[b+3]

  if m=='u':
    temp[b-3], temp[b]=temp[b], temp[b-3]

  if m=='l':
    temp[b-1],temp[b]=temp[b], temp[b-1]

  if m=='r':
    temp[b+1],temp[b]=temp[b],temp[b+1]

  return temp

src=[1,2,3,4,5,6,0,7,8]
target=[1,2,3,4,5,6,7,8,0]
bfs(src, target)
```

```
[1, 2, 3, 4, 5, 6, 0, 7, 8]
[1, 2, 3, 0, 5, 6, 4, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[0, 2, 3, 1, 5, 6, 4, 7, 8]
[1, 2, 3, 5, 0, 6, 4, 7, 8]
[1, 2, 3, 4, 0, 6, 7, 5, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
success
```

```python
# water jug problem using BFS
from collections import deque
# Function to find all possible states from the current state
def find_states(state, capacities):
    a, b = state
    a_capacity, b_capacity = capacities
    states = []

    # All possible operations: Fill, Empty, Pour
    operations = [
        (a_capacity, b),   # Fill jug A
        (a, b_capacity),   # Fill jug B
        (0, b),            # Empty jug A
        (a, 0),            # Empty jug B
        (min(a + b, a_capacity), max(0, a + b - a_capacity)),  # Pour from B to A
        (max(0, a + b - b_capacity), min(a + b, b_capacity))   # Pour from A to B
    ]

    for operation in operations:
        if operation != state:  # Avoid adding the current state
            states.append(operation)
    return states

# Function to perform Breadth-First Search
def bfs(start_state, target, capacities):
    visited = set()
    queue = deque([(start_state, [])])  # Initialize queue with start state and empty path

    while queue:
        current_state, path = queue.popleft()

        if current_state == target:
            return path + [current_state]

        if current_state not in visited:
            visited.add(current_state)
            next_states = find_states(current_state, capacities)

            for next_state in next_states:
                queue.append((next_state, path + [current_state]))

    return None

start_state = (0, 0)  # Initial state of jugs (jug A, jug B)
target_state = (2, 0)  # Target state to achieve (2 units in jug A)
jug_capacities = (4, 3)  # Capacities of the jugs (jug A capacity, jug B capacity)

result = bfs(start_state, target_state, jug_capacities)
if result:
    print("Path to reach the target state:", result)
else:
    print("Target state cannot be reached from the given start state.")
```

```
Path to reach the target state: [(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]
```

```python
# water jug problem using DFS with visited set
def solveWaterJugProblem(capacity_jug1, capacity_jug2, desired_quantity):
    stack = []
    visited = set()  # Visited set to store explored states
    stack.append((0, 0))  # Initial state: both jugs empty
    visited.add((0, 0))
    while stack:
        current_state = stack.pop()
        if current_state[0] == desired_quantity or current_state[1] == desired_quantity:
            return current_state
        next_states = generateNextStates(current_state, capacity_jug1, capacity_jug2)
        for state in next_states:
            if state not in visited:
                stack.append(state)
                visited.add(state)
    return "No solution found"

def generateNextStates(state, capacity_jug1, capacity_jug2):
    next_states = []
    # Fill Jug 1
    next_states.append((capacity_jug1, state[1]))
    # Fill Jug 2
    next_states.append((state[0], capacity_jug2))
    # Empty Jug 1
    next_states.append((0, state[1]))
    # Empty Jug 2
    next_states.append((state[0], 0))
    # Pour water from Jug 1 to Jug 2
    pour_amount = min(state[0], capacity_jug2 - state[1])
    next_states.append((state[0] - pour_amount, state[1] + pour_amount))
    # Pour water from Jug 2 to Jug 1
    pour_amount = min(state[1], capacity_jug1 - state[0])
    next_states.append((state[0] + pour_amount, state[1] - pour_amount))
    return next_states

# Driver Code
if __name__ == "__main__":
    capacity_jug1 = 5
    capacity_jug2 = 3
    desired_quantity = 4
    result = solveWaterJugProblem(capacity_jug1, capacity_jug2, desired_quantity)
    print("Final state:", result)
```

```
Final state: (4, 0)
```

```python
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, move=None, depth=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = self.calculate_cost()

    def __lt__(self, other):
        return self.cost < other.cost

    def calculate_cost(self):
        return self.depth + self.heuristic()

    def heuristic(self):
        # Manhattan distance heuristic
        h = 0
        goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        for i in range(1, 9):
            s_row, s_col = self.state.index(i) // 3, self.state.index(i) % 3
            g_row, g_col = goal_state.index(i) // 3, goal_state.index(i) % 3
            h += abs(s_row - g_row) + abs(s_col - g_col)
        return h

    def expand_node(self):
        successors = []
        zero_index = self.state.index(0)
        next_moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # Up, Down, Left, Right
        for dx, dy in next_moves:
            new_row, new_col = zero_index // 3 + dx, zero_index % 3 + dy
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = self.state[:]
                new_index = new_row * 3 + new_col
                new_state[zero_index], new_state[new_index] = new_state[new_index], new_state[zero_index]
                successors.append(PuzzleNode(new_state, self, f"Move {new_state[zero_index]} to {zero_index}", self.depth + 1))
        return successors

    def is_goal_state(self):
        return self.state == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_solution(self):
        solution = []
        node = self
        while node:
            solution.append((node.move, node.state))
            node = node.parent
        solution.reverse()
        return solution

def astar(initial_state):
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, initial_state)
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.is_goal_state():
            return current_node.get_solution()
        closed_list.add(tuple(current_node.state))
        successors = current_node.expand_node()
        for successor in successors:
            if tuple(successor.state) not in closed_list:
                heapq.heappush(open_list, successor)
    return None

if __name__ == "__main__":
    initial_state = [2, 8, 3, 1, 6, 4, 7, 0, 5]  # Example initial state
    initial_node = PuzzleNode(initial_state)
    solution = astar(initial_node)
    if solution:
        print("Solution found!")
        for move, state in solution:
            print(move, state)
    else:
        print("No solution found!")
```

```
    No solution found!
```

```python
# tic-tac-toe using minimax
import math

class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.current_winner = None

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def num_empty_squares(self):
        return self.board.count(' ')

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def winner(self, square, letter):
        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([spot == letter for spot in row]):
            return True

        col_ind = square % 3
        col = [self.board[col_ind+i*3] for i in range(3)]
        if all([spot == letter for spot in col]):
            return True

        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            if all([spot == letter for spot in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all([spot == letter for spot in diagonal2]):
                return True
        return False

    def minimax(self, board, depth, max_player):
        if self.current_winner == 'X':
            return {'position': None, 'score': 1 * (self.num_empty_squares() + 1) if max_player else -1 * (self.num_empty_squares() + 1)}
        elif self.current_winner == 'O':
            return {'position': None, 'score': -1 * (self.num_empty_squares() + 1) if max_player else 1 * (self.num_empty_squares() + 1)}
        elif self.num_empty_squares() == 0:
            return {'position': None, 'score': 0}

        if max_player:
            best = {'position': None, 'score': -math.inf}
            for possible_move in self.available_moves():
                board.make_move(possible_move, 'X')
                sim_score = self.minimax(board, depth - 1, False)
                board.board[possible_move] = ' '
                sim_score['position'] = possible_move
                if sim_score['score'] > best['score']:
                    best = sim_score
        else:
            best = {'position': None, 'score': math.inf}
            for possible_move in self.available_moves():
                board.make_move(possible_move, 'O')
                sim_score = self.minimax(board, depth - 1, True)
                board.board[possible_move] = ' '
                sim_score['position'] = possible_move
                if sim_score['score'] < best['score']:
                    best = sim_score
        return best
```

```python
if __name__ == "__main__":
    game = TicTacToe()
    game.print_board()
    print("Sample Input: Choose the cell number to make your move (0-8)")
    print("Sample Output: The updated board with your move")
    while game.num_empty_squares() > 0 and not game.current_winner:
        try:
            square = int(input("Your move (X): "))
            game.make_move(square, 'X')
            game.print_board()
            if game.current_winner:
                print("You win!")
                break
            move = game.minimax(game, game.num_empty_squares(), False)
            game.make_move(move['position'], 'O')
            print("AI move (O):", move['position'])
            game.print_board()
            if game.current_winner:
                print("AI wins!")
                break
        except ValueError:
            print("Invalid move. Please enter a number between 0 and 8.")
#Sample input 4
```

```
|   |   |   |
|   |   |   |
|   |   |   |
Sample Input: Choose the cell number to make your move (0-8)
Sample Output: The updated board with your move
Your move (X): 4
|   |   |   |
|   | X |   |
|   |   |   |
AI move (O): 0
| O |   |   |
|   | X |   |
|   |   |   |
AI wins!
```

```
# graph coloring problem
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0 for _ in range(vertices)] for _ in range(vertices)]

    def add_edge(self, u, v):
        self.graph[u][v] = 1
        self.graph[v][u] = 1

    def is_safe(self, v, color, c):
        for i in range(self.vertices):
            if self.graph[v][i] == 1 and color[i] == c:
```

```
# knapsack problem brute force
def knapSack(W, wt, val, n):
    # Base case: If no items left or knapsack capacity is 0, return 0
    if n == 0 or W == 0:
        return 0

    # If weight of nth item > Knapsack capacity W, item cannot be included in the optimal solution
    if wt[n-1] > W:
        return knapSack(W, wt, val, n-1)

    # Return the maximum of two cases: nth item included and not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1), knapSack(W, wt, val, n-1))

# Driver Code
if __name__ == '__main__':
     profit = [60, 100, 120]
     weight = [10, 20, 30]
     W = 50
     n = len(profit)
     print(knapSack(W, weight, profit, n))
```

```
220
```