```python
graph = {
    'S': ['A', 'B'],   #list[] to store the nodes
    'A':['C','D'],
    'B':['G','H'],
    'C':['E','F'],
    'D':[],
    'E':['K'],
    'F':[],
    'G':['I'],
    'H':[],
    'I':[],
    'K':[],
}

visited=[] #List of visited nodes.
queue=[]   #initialization of queue

def bfs(visited,graph,node):
    visited.append(node)
    queue.append(node)

    while queue:
        m=queue.pop(0)
        print(m , end=" ")
        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

print("The Result of Breadth First Search is as follows: ")
bfs(visited,graph,'S')
```

```python
graph = {
    '3':['5','8','25'],
    '5':['1','2'],
    '8':[],
    '25':['12','8'],
    '1':[],
    '2':[],
    '12':['6'],
    '6' :['4','9'],
    '4':[],
    '9':[]
    }

visited = set()
def dfs(visited, graph, node):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited,graph,neighbour)

dfs(visited,graph,'3')
```

```python
def TowerofHanoi(n,s_pole,d_pole,i_pole):
    if n==1:
        print("Move Disc 1 From Pole",s_pole,"to pole",d_pole)
        return
    TowerofHanoi(n-1,s_pole,i_pole,d_pole)
    print("Move Disc",n,"from pole",s_pole,"to pole",d_pole)
    TowerofHanoi(n-1,i_pole,d_pole,s_pole)

n=3
TowerofHanoi(n,'A','C','B')
```

```python
print("Enter the number of queens")
N = int(input())
# chessboard
# NxN matrix with all elements 0
board = [[0] * N for _ in range(N)]

def is_attack(i, j):
    # check if there is a queen in row or column
    for k in range(0, N):
        if board[i][k] == 1 or board[k][j] == 1:
            return True
    # checking diagonals
    for k in range(0, N):
        for l in range(0, N):
            if (k + l == i + j) or (k - l == i - j):
                if board[k][l] == 1:
                    return True
    return False

def N_queen(n):
    # if n is 0, solution found
    if n == 0:
        return True

    for i in range(0, N):
        for j in range(0, N):
            if (not is_attack(i, j)) and (board[i][j] != 1):
                board[i][j] = 1
                if N_queen(n - 1):
                    return True
                board[i][j] = 0
    return False


if N_queen(N):
    for i in board:
        print(i)
else:
    print("Solution does not exist.")
```

```python
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]
    if maximizingPlayer:
        best = MIN
        # Recur for left & right children
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            # Alpha-Beta Pruning
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        # Recur for left and right children
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            # Alpha-Beta Pruning
            if beta <= alpha:
                break
        return best

# Driver Code
if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

```
capacity = (12, 8, 5)
# Maximum capacities of 3 jugs -->x,y,z
x = capacity[0]
y = capacity[1]
z = capacity[2]
# to mark visited states
memory = {}
# store solution path
ans = []

def get_all_states(state):
    # Let the 3 jugs be called a,b,c
    a = state[0]
    b = state[1]
    c = state[2]

    if (a == 6 and b == 6):
        ans.append(state)
        return True
    # if current state is already visited earlier
    if ((a, b, c) in memory):
        return False
    memory[(a,b,c)]=1
    #empty jug a
    if a>0:
        #empty a into b
        if(a+b<=y):
            if(get_all_states((0,a+b,c))):
                ans.append(state)
                return True
        else:
            if (get_all_states((a-(y-b),y,c))):
                ans.append(state)
                return True

        #empty a into c
        if (a+c<=z):
            if(get_all_states((0,b,a+c))):
                ans.append(state)
                return True
        else:
            if (get_all_states((a-(z-c),b,z))):
                ans.append(state)
                return True

    #empty jug b
    if (b>0):
        #empty b into a
        if (a + b <= x):
            if (get_all_states((a + b, 0, c))):
                ans.append(state)
                return True
        else:
            if (get_all_states((x, b - (x - a), c))):
                ans.append(state)
                return True
        #empty b into c
        if(b+c<=z):
            if(get_all_states((a,0,b+c))):
                ans.append(state)
                return True
        else:
            if(get_all_states((a,b-(z-c),z))):
                ans.append(state)
                return True

    #empty jug c
    if(c>0):
        #empty c into a
        if(a+c<=x):
            if(get_all_states((a+c,b,0))):
                ans.append(state)
                return True
        else:
            if(get_all_states((x,b,c-(x-a)))):
                ans.append(state)
                return True
        #empty c into b
        if (b + c <= y):
            if (get_all_states((a, b + c, 0))):
                ans.append(state)
                return True
        else:
            if (get_all_states((a, y, c - (y - b)))):
                ans.append(state)
                return True
    return False
```

```python
initial_state=(12,0,0)
print("starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

```python
from collections import deque

initial_state = (3, 3, 1)
goal_state = (0, 0, 0)

def is_valid(state):
    missionaries_left, cannibals_left, boat_location = state

    if (
        missionaries_left < 0
        or cannibals_left < 0
        or missionaries_left > 3
        or cannibals_left > 3
    ):
        return False

    if (
        (missionaries_left < cannibals_left and missionaries_left > 0)
        or (3 - missionaries_left < 3 - cannibals_left and 3 - missionaries_left > 0)
    ):
        return False

    return True

def get_next_states(state):
    states = []
    missionaries_left, cannibals_left, boat_location = state

    passengers = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

    for m, c in passengers:
        new_missionaries_left = missionaries_left - m * boat_location
        new_cannibals_left = cannibals_left - c * boat_location
        new_boat_location = 1 - boat_location

        new_state = (new_missionaries_left, new_cannibals_left, new_boat_location)

        if is_valid(new_state):
            states.append(new_state)

    return states

def solve():
    visited = set()
    queue = deque([(initial_state, [])])

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

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

        visited.add(current_state)
        next_states = get_next_states(current_state)

        for next_state in next_states:
            if next_state not in visited:
                queue.append((next_state, path + [current_state]))

    return None

solution = solve()

if solution:
    print("Solution:")
    for i, state in enumerate(solution):
        print(f"Step {i + 1}: {state}")
else:
    print("No solution found")
```

```python
import random

ranks =['2','3','4','5','6','7','8','9','10','jack','queen','king','ace']
suits=['hearts','diamonds','clubs','spades']
deck=[{'rank':rank,'suit':suit} for rank in ranks for suit in suits]

def shuffle_deck(deck):
    random.shuffle(deck)

def display_deck(deck):
    for card in deck:
        print(f"{card['rank']} of {card['suit']}")

shuffle_deck(deck)
print("shuffles deck:")
display_deck(deck)


class BlockWorld:
    def __init__(self):
        self.state = ['A', 'B', 'C'] #Initial state: Three blocks on table

    def move(self, source, destination):

        if source == destination or source not in self.state:
            return False #Invalid move

        block = self.state.pop(self.state.index(source))  #Remove block from source

        self.state.append(block)  #place the block on destination
        return True

    def solve(self):
        #Move A to C
        self.move('A','C')

        #Move A to B
        self.move('A','B')

        #Move C to B
        self.move('C','B')

        #Move A to C
        self.move('A','C')

        #Move B to A
        self.move('B','A')

        #Move B to C
        self.move('B','C')

        #Move A to C
        self.move('A','C')

        return self.state

# Create a BlockWorld instance
block_world = BlockWorld()

#Solve the problem
final_state = block_world.solve()

print("Final state:", final_state)


from simpleai.search import SearchProblem, astar
GOAL = 'HELLO WORLD'
class HelloProblem(SearchProblem):
    def actions(self, state):
        if len(state) < len(GOAL):
            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        else:
            return []

    def result(self, state, action):
        return state + action
    def is_goal(self, state):
        return state == GOAL

    def heuristic(self, state):
        wrong = sum([1 if state[i] != GOAL[i] else 0 for i in range(len(state))])
        missing = len(GOAL) - len(state)
        return wrong + missing
problem = HelloProblem(initial_state='')
result = astar(problem)
print(result.state)
print(result.path())
```