Experiment-2

Aim - 8 Puzzle Single Player Game (Breadth First Search)

Code-

```
# Import the necessary libraries
from time import time
from queue import Queue
# Creating a class Puzzle
class Puzzle:
    # Setting the goal state of 8-puzzle
    goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]
    num_of_instances = 0
    # constructor to initialize the class members
    def __init__(self, state, parent, action):
        self.parent = parent
        self.state = state
        self.action = action
        # Incrementing the number of instances by 1
        Puzzle.num_of_instances += 1
    # function used to display a state of 8-puzzle
    def __str__(self):
        return str(self.state[0:3]) + '\n' + str(self.state[3:6]) + '\n' + str(self.state[6:9])
    # method to compare the current state with the goal state
    def goal_test(self):
        # Comparing the current state with the goal state
        if self.state == Puzzle.goal_state:
            return True
        return False
```

```
# static method to find the legal action based on the current board position
    @staticmethod
    def find_legal_actions(i, j):
        legal_action = ['U', 'D', 'L', 'R']
        if i == 0:
            # if row is 0 in board, then 'U' (Up) is disabled
            legal_action.remove('U')
        elif i == 2:
            # if row is 2 in board, then 'D' (Down) is disabled
            legal_action.remove('D')
        if j == 0:
            # if column is 0, then 'L' (Left) is disabled
            legal_action.remove('L')
        elif j == 2:
            # if column is 2, then 'R' (Right) is disabled
            legal_action.remove('R')
        return legal_action
    # method to generate the child of the current state of the board
    def generate_child(self):
        # Create an empty list for children
        children = []
        x = self.state.index(0)
        i = int(x / 3)
        j = int(x % 3)
        # Find the legal actions based on i and j values
        legal_actions = self.find_legal_actions(i, j)
        # Iterate over all legal actions
        for action in legal_actions:
            new_state = self.state.copy()
```

```
            # If the legal action is UP
            if action == 'U':
                # Swapping between current index of 0 with its up element on the board
                new_state[x], new_state[x - 3] = new_state[x - 3], new_state[x]
            elif action == 'D':
                # Swapping between current index of 0 with its down element on the board
                new_state[x], new_state[x + 3] = new_state[x + 3], new_state[x]
            elif action == 'L':
                # Swapping between the current index of 0 with its left element on the board
                new_state[x], new_state[x - 1] = new_state[x - 1], new_state[x]
            elif action == 'R':
                # Swapping between the current index of 0 with its right element on the board
                new_state[x], new_state[x + 1] = new_state[x + 1], new_state[x]
            children.append(Puzzle(new_state, self, action))
        # Return the children
        return children
    # method to find the solution
    def find_solution(self):
        solution = []
        solution.append(self.action)
        path = self
        while path.parent != None:
            path = path.parent
            solution.append(path.action)
        solution = solution[:-1]
        solution.reverse()
        return solution
# method for breadth first search
def breadth_first_search(initial_state):
```

```
    start_node = Puzzle(initial_state, None, None)
    print("Initial state:")
    print(start_node)
    if start_node.goal_test():
        return start_node.find_solution()
    q = Queue()
    q.put(start_node)
    explored = []
    # Iterate the queue until empty
    while not q.empty():
        node = q.get()
        # Append the state of node in the explored list
        explored.append(node.state)
        # Generate the child nodes of the current node
        children = node.generate_child()
        # Iterate over each child node in children
        for child in children:
            if child.state not in explored:
                if child.goal_test():
                    return child.find_solution()
                q.put(child)
    return None
# Start executing the 8-puzzle with setting up the initial state
state = [
    [1, 3, 4, 8, 6, 2, 7, 0, 5],
    [2, 8, 1, 0, 4, 3, 7, 6, 5],
    [2, 8, 1, 4, 6, 3, 0, 7, 5]
]
```

```
# Iterate over number of initial states
for i in range(0, 3):
    # Initialize the num_of_instances to zero
    Puzzle.num_of_instances = 0
    # Set t0 to current time
    t0 = time()
    # Call breadth_first_search
    bfs = breadth_first_search(state[i])
    # Get the time t1 after executing the breadth_first_search method
    t1 = time() - t0
    # Output the result of BFS, the space used (number of instances created), and the time taken
    print('BFS Solution:', bfs)
    print('Space (number of instances):', Puzzle.num_of_instances)
    print('Time taken:', t1)
    print()
print('------------------------------------------')
```

Output –

```
Initial state:
[1, 3, 4]
[8, 6, 2]
[7, 0, 5]
BFS Solution: ['U', 'R', 'U', 'L', 'D']
Space (number of instances): 66
Time taken: 0.0

Initial state:
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
BFS Solution: ['U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Space (number of instances): 591
Time taken: 0.003746509552001953

Initial state:
[2, 8, 1]
[4, 6, 3]
[0, 7, 5]
BFS Solution: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Space (number of instances): 2956
Time taken: 0.04059553146362305

------------------------------------------
```

Experiment-3

Aim -8 Puzzle Single Player Game (A* Algorithm)

Code-

```
from time import time
from queue import PriorityQueue
import math
class Puzzle:
    goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]
    heuristic = None
    evaluation_function = None
    needs_heuristic = False
    num_of_instances = 0
    def __init__(self, state, parent, action, path_cost, needs_heuristic=False):
        self.parent = parent
        self.state = state
        self.action = action
        if parent:
            self.path_cost = parent.path_cost + path_cost
        else:
            self.path_cost = path_cost
        if needs_heuristic:
            self.needs_heuristic = True
            self.generate_heuristic()
            self.evaluation_function = self.path_cost + self.heuristic
        else:
            self.evaluation_function = self.path_cost
        Puzzle.num_of_instances += 1
```

```
def __str__(self):
    return str(self.state[0:3]) + '\n' + str(self.state[3:6]) + '\n' + str(self.state[6:9])
def generate_heuristic(self):
    self.heuristic = 0
    for num in range(1, 9):
        distance = self.state.index(num)
        goal_index = Puzzle.goal_state.index(num)
        i = int(distance / 3)
        j = int(distance % 3)
        goal_i = int(goal_index / 3)
        goal_j = int(goal_index % 3)
        self.heuristic += abs(i - goal_i) + abs(j - goal_j)
def goal_test(self):
    if self.state == Puzzle.goal_state:
        return True
    return False
@staticmethod
def find_legal_actions(i, j):
    legal_action = ['U', 'D', 'L', 'R']
    if i == 0:
        legal_action.remove('U')
    elif i == 2:
        legal_action.remove('D')
    if j == 0:
        legal_action.remove('L')
    elif j == 2:
        legal_action.remove('R')
    return legal_action
def generate_child(self):
```

```
        children = []
        x = self.state.index(0)
        i = x // 3
        j = x % 3
        legal_actions = Puzzle.find_legal_actions(i, j)
        for action in legal_actions:
            new_state = self.state.copy()
            if action == 'U':
                new_state[x], new_state[x - 3] = new_state[x - 3], new_state[x]
            elif action == 'D':
                new_state[x], new_state[x + 3] = new_state[x + 3], new_state[x]
            elif action == 'L':
                new_state[x], new_state[x - 1] = new_state[x - 1], new_state[x]
            elif action == 'R':
                new_state[x], new_state[x + 1] = new_state[x + 1], new_state[x]
            children.append(Puzzle(new_state, self, action, 1, True))
        return children
    def find_solution(self):
        solution = []
        solution.append(self.action)
        path = self
        while path.parent is not None:
            path = path.parent
            solution.append(path.action)
        solution = solution[:-1]
        solution.reverse()
        return solution
def Astar_search(initial_state):
    count = 0
```

```
    explored = []
    start_node = Puzzle(initial_state, None, None, 0, True)
    q = PriorityQueue()
    q.put((start_node.evaluation_function, count, start_node))
    while not q.empty():
        _, _, node = q.get()
        explored.append(node.state)
        if node.goal_test():
            return node.find_solution()
        children = node.generate_child()
        for child in children:
            if child.state not in explored:
                count += 1
                q.put((child.evaluation_function, count, child))
    return None
state = [
    [1, 3, 4, 8, 6, 2, 7, 0, 5],
    [2, 8, 1, 0, 4, 3, 7, 6, 5],
    [2, 8, 1, 4, 6, 3, 0, 7, 5]
]
for i in range(0, 3):
    Puzzle.num_of_instances = 0
    t0 = time()
    astar = Astar_search(state[i])
    t1 = time() - t0
    print('A* Solution:', astar)
    print('Space (number of instances):', Puzzle.num_of_instances)
    print('Time taken:', t1)
    print()
```

```python
print('-----------------------------------------')
```

Output –

```
A* Solution: ['U', 'R', 'U', 'L', 'D']
Space (number of instances): 16
Time taken: 0.0

A* Solution: ['U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Space (number of instances): 42
Time taken: 0.0009996891021728516

A* Solution: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Space (number of instances): 71
Time taken: 0.0009999275207519531

------------------------------------------
```

Experiment-4

Aim- WATER JUG PROBLEM USING BFS & DFS

Code -

```
import collections


def get_index(node):
    return pow(7, node[0]) * pow(5, node[1])


def get_search_type():
    s = input("Enter 'b' for BFS, 'd' for DFS: ")
    s = s.lower()

    while s != 'b' and s != 'd':
        s = input("The input is not valid! Enter 'b' for BFS, 'd' for DFS: ").lower()

    return True if s == 'b' else False


def get_jugs():
    print("Receiving the volume of the jugs...")
    jugs = []

    temp = int(input("Enter first jug volume (>1): "))
    while temp < 1:
        temp = int(input("Enter a valid amount (>1): "))
    jugs.append(temp)

    temp = int(input("Enter second jug volume (>1): "))
```

```
    while temp < 1:
        temp = int(input("Enter a valid amount (>1): "))
    jugs.append(temp)

    return jugs

def get_goal(jugs):
    print("Receiving the desired amount of the water...")
    max_amount = max(jugs)
    s = "Enter the desired amount of water (1 - {0}): ".format(max_amount)
    goal_amount = int(input(s))

    while goal_amount < 1 or goal_amount > max_amount:
        goal_amount = int(input("Enter a valid amount (1 - {0}): ".format(max_amount)))

    return goal_amount

def is_goal(path, goal_amount):
    print("Checking if the goal is achieved...")
    return path[-1][0] == goal_amount or path[-1][1] == goal_amount

def been_there(node, check_dict):
    print("Checking if {0} is visited before...".format(node))
    return check_dict.get(tuple(node), False)

def next_transitions(jugs, path, check_dict):
    print("Finding next transitions and checking for the loops...")
    result = []
    next_nodes = []
```

```
node = []

a_max = jugs[0]
b_max = jugs[1]

a = path[-1][0]
b = path[-1][1]

node.append(a_max)
node.append(b)
if not been_there(node, check_dict):
    next_nodes.append(node)
node = []

node.append(a)
node.append(b_max)
if not been_there(node, check_dict):
    next_nodes.append(node)
node = []

node.append(min(a_max, a + b))
node.append(b - (node[0] - a))
if not been_there(node, check_dict):
    next_nodes.append(node)
node = []

node.append(a - (min(a + b, b_max) - b))
node.insert(0, min(a + b, a_max))
if not been_there(node, check_dict):
```

```
        next_nodes.append(node)
    node = []

    node.append(0)
    node.append(b)
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    node.append(a)
    node.append(0)
    if not been_there(node, check_dict):
        next_nodes.append(node)

    for i in range(0, len(next_nodes)):
        temp = list(path)
        temp.append(next_nodes[i])
        result.append(temp)

    if len(next_nodes) == 0:
        print("No more unvisited nodes...\nBacktracking...")
    else:
        print("Possible transitions: ")
        for nnode in next_nodes:
            print(nnode)

    return result

def transition(old, new, jugs):
```

```
    a = old[0]
    b = old[1]
    a_prime = new[0]
    b_prime = new[1]
    a_max = jugs[0]
    b_max = jugs[1]

    if a > a_prime:
        if b == b_prime:
            return "Clear {0}-liter jug:\t\t\t".format(a_max)
        else:
            return "Pour {0}-liter jug into {1}-liter jug:\t".format(a_max, b_max)
    else:
        if b > b_prime:
            if a == a_prime:
                return "Clear {0}-liter jug:\t\t\t".format(b_max)
            else:
                return "Pour {0}-liter jug into {1}-liter jug:\t".format(b_max, a_max)
        else:
            if a == a_prime:
                return "Fill {0}-liter jug:\t\t\t".format(b_max)
            else:
                return "Fill {0}-liter jug:\t\t\t".format(a_max)


def print_path(path, jugs):
    print("Starting from:\t\t\t\t", path[0])
    for i in  range(0, len(path) - 1):
        print(i + 1, ":", transition(path[i], path[i + 1], jugs), path[i + 1])
```

```
def search(starting_node, jugs, goal_amount, check_dict, is_breadth):
    if is_breadth:
        print("Implementing BFS...")
    else:
        print("Implementing DFS...")

    goal = []
    accomplished = False
    q = collections.deque()
    q.appendleft(starting_node)

    while len(q) != 0:
        path = q.popleft()
        check_dict[get_index(path[-1])] = True
        if len(path) >= 2:
            print(transition(path[-2], path[-1], jugs), path[-1])
        if is_goal(path, goal_amount):
            accomplished = True
            goal = path
            break

        next_moves = next_transitions(jugs, path, check_dict)
        for i in next_moves:
            if is_breadth:
                q.append(i)
            else:
                q.appendleft(i)

    if accomplished:
```

```
        print("The goal is achieved\nPrinting the sequence of the moves...\n")
        print_path(goal, jugs)
    else:
        print("Problem cannot be solved.")


if __name__ == '__main__':
    starting_node = [[0, 0]]
    jugs = get_jugs()
    goal_amount = get_goal(jugs)
    check_dict = {}
    is_breadth = get_search_type()
    search(starting_node, jugs, goal_amount, check_dict, is_breadth)
```

Experiment-5

Aim-Tic-Tac-Toe Game using Min-Max Algorithm

Code-

```
import numpy as np
from math import inf as infinity
# Set the Empty Board
game_state = [[' ',' ',' '],
              [' ',' ',' '],
              [' ',' ',' ']]
# Create the Two Players as 'X'/'O'
players = ['X', 'O']
# Method for checking the correct move on Tic-Tac-Toe
def play_move(state, player, block_num):
    if state[int((block_num-1)/3)][(block_num-1)%3] == ' ':
        state[int((block_num-1)/3)][(block_num-1)%3] = player
    else:
        block_num = int(input("Block is not empty, ya blockhead! Choose again: "))
        play_move(state, player, block_num)
# Method to copy the current game state to new_state of Tic-Tac-Toe
def copy_game_state(state):
    new_state = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]
    for i in range(3):
        for j in range(3):
            new_state[i][j] = state[i][j]
    return new_state
# Method to check the current state of the Tic-Tac-Toe
def check_current_state(game_state):
```

```
    draw_flag = 1
    for i in range(3):
        for j in range(3):
            if game_state[i][j] == ' ':
                draw_flag = 0
    if draw_flag == 1:
        return None, "Draw"
    # Check horizontals
    for i in range(3):
        if game_state[i][0] == game_state[i][1] == game_state[i][2] and game_state[i][0] != ' ':
            return game_state[i][0], "Done"
    # Check verticals
    for j in range(3):
        if game_state[0][j] == game_state[1][j] == game_state[2][j] and game_state[0][j] != ' ':
            return game_state[0][j], "Done"
    # Check diagonals
    if game_state[0][0] == game_state[1][1] == game_state[2][2] and game_state[0][0] != ' ':
        return game_state[0][0], "Done"
    if game_state[0][2] == game_state[1][1] == game_state[2][0] and game_state[0][2] != ' ':
        return game_state[0][2], "Done"

    return None, "Not Done"
# Method to print the Tic-Tac-Toe Board
def print_board(game_state):
    print('----------------')
    for row in game_state:
        print('| ' + ' || '.join(row) + ' |')
        print('----------------')
# Method for implementing the Minimax Algorithm
```

```
def getBestMove(state, player):
    winner_loser, done = check_current_state(state)
    if done == "Done" and winner_loser == 'O':
        return 1
    elif done == "Done" and winner_loser == 'X':
        return -1
    elif done == "Draw":
        return 0
    moves = []
    empty_cells = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == ' ':
                empty_cells.append(i*3 + (j+1))

    for empty_cell in empty_cells:
        move = {}
        move['index'] = empty_cell
        # Copy the game state
        new_state = copy_game_state(state)
        # Simulate the move
        play_move(new_state, player, empty_cell)
        if player == 'O':
            result = getBestMove(new_state, 'X')
            move['score'] = result
        else:
            result = getBestMove(new_state, 'O')
            move['score'] = resul
        moves.append(move)
```

```
    # Find best move
    best_move = None
    if player == "O":  # Computer's turn
        best = -infinity
        for move in moves:
            if move['score'] > best:
                best = move['score']
                best_move = move['index']
    else:  # Human's turn
        best = infinity
        for move in moves:
            if move['score'] < best:
                best = move['score']
                best_move = move['index']
    return best_move
# Now Playing the Tic-Tac-Toe Game
play_again = 'Y'
while play_again == 'Y' or play_again == 'y':
    game_state = [[' ',' ',' '],
                  [' ',' ',' '],
                  [' ',' ',' ']]
    current_state = "Not Done"
    print("\nNew Game!")
    print_board(game_state)
    player_choice = input("Choose which player goes first - X (You) or O(Computer): ")
    winner = None
    if player_choice == 'X' or player_choice == 'x':
        current_player_idx = 0
    else:
```

```
    current_player_idx = 1
  while current_state == "Not Done":
    if current_player_idx == 0:  # Human's turn
      block_choice = int(input("Your turn please! Choose where to place (1 to 9): "))
      play_move(game_state, players[current_player_idx], block_choice)
    else:  # Computer's turn
      block_choice = getBestMove(game_state, players[current_player_idx])
      play_move(game_state, players[current_player_idx], block_choice)
      print("AI plays move: " + str(block_choice))
    print_board(game_state)
    winner, current_state = check_current_state(game_state)
    if winner is not None:
      print(str(winner) + " won!")
    else:
      current_player_idx = (current_player_idx + 1) % 2
    if current_state == "Draw":
      print("Draw!")
  play_again = input('Wanna try again?(Y/N) : ')
  if play_again == 'N':
    print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')
```

Output-

```
New Game!
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Choose which player goes first - X (You) or O(Computer):  x
Your turn please! Choose where to place (1 to 9):  1
```

```
----------------
| X ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
AI plays move: 2
----------------
| X || O ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9):  5
----------------
| X || O ||   |
----------------
|   || X ||   |
----------------
|   ||   ||   |
----------------
AI plays move: 3
----------------
| X || O || O |
----------------
|   || X ||   |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9):  9
----------------
| X || O || O |
----------------
|   || X ||   |
----------------
|   ||   || X |
----------------
X won!
```

Experiment-7

Aim - Constraint Satisfaction Problems

Code-

```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod
V = TypeVar('V')
D = TypeVar('D')
class Constraint(Generic[V, D], ABC):
    def __init__(self, variables: List[V]) -> None:
        self.variables = variables
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        pass
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        self.variables: List[V] = variables
        self.domains: Dict[V, List[D]] = domains
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        for variable in self.variables:
            self.constraints[variable] = []
            if variable not in self.domains:
                raise LookupError("Every variable should have a domain assigned to it.")
    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Variable in constraint not in CSP")
            else:
                self.constraints[variable].append(constraint)
```

```
    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        for constraint in self.constraints[variable]:
            if not constraint.satisfied(assignment):
                return False
        return True
    def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
        if len(assignment) == len(self.variables):
            return assignment
        unassigned: List[V] = [v for v in self.variables if v not in assignment]
        first: V = unassigned[0]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            local_assignment[first] = value
            if self.consistent(first, local_assignment):
                result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
                if result is not None:
                    return result
        return None
class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2
    def satisfied(self, assignment: Dict[str, str]) -> bool:
        if self.place1 not in assignment or self.place2 not in assignment:
            return True
        return assignment[self.place1] != assignment[self.place2]
if __name__ == "__main__":
    variables: List[str] = ["BOX_1", "BOX_2", "BOX_4", "BOX_3", "BOX_5", "BOX_6", "BOX_7"]
```

```
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["red", "green", "blue"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("BOX_1", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_1", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_4", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_5"))
    csp.add_constraint(MapColoringConstraint("BOX_5", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_5"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_7"))
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
class SendMoreMoneyConstraint(Constraint[str, int]):
    def __init__(self, letters: List[str]) -> None:
        super().__init__(letters)
        self.letters: List[str] = letters
    def satisfied(self, assignment: Dict[str, int]) -> bool:
        if len(set(assignment.values())) < len(assignment):
            return False
        if len(assignment) == len(self.letters):
            s: int = assignment["S"]
            e: int = assignment["E"]
```

```
            n: int = assignment["N"]
            d: int = assignment["D"]
            m: int = assignment["M"]
            o: int = assignment["O"]
            r: int = assignment["R"]
            y: int = assignment["Y"]
            send: int = s * 1000 + e * 100 + n * 10 + d
            more: int = m * 1000 + o * 100 + r * 10 + e
            money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
            return send + more == money
        return True
if __name__ == "__main__":
    letters: List[str] = ["S", "E", "N", "D", "M", "O", "R", "Y"]
    possible_digits: Dict[str, List[int]] = {}
    for letter in letters:
        possible_digits[letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    possible_digits["M"] = [1]
    csp: CSP[str, int] = CSP(letters, possible_digits)
    csp.add_constraint(SendMoreMoneyConstraint(letters))
    solution: Optional[Dict[str, int]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

Output-

```
{'BOX_1': 'red', 'BOX_2': 'green', 'BOX_4': 'blue', 'BOX_3': 'red', 'BOX_5': 'green', 'BOX_6': 'red', 'BOX_7': 'green'}
{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}
```

Experiment-8

Aim-Implement a Knapsack problem using Brute Force Method and Dynamic Programming

Code-

```
from itertools import product

from collections import namedtuple

try:

    from itertools import izip

except ImportError:

    izip = zip

    Reward = namedtuple('Reward', 'name value weight volume')

bagpack = Reward('bagpack', 0, 25.0, 0.25)

items = [Reward('laptop', 3000, 0.3, 0.025), Reward('printer', 1800, 0.2, 0.015), Reward('headphone', 
2500, 2.0, 0.002)]

def tot_value(items_count):

    global items, bagpack

    weight = sum(n * item.weight for n, item in izip(items_count, items))

    volume = sum(n * item.volume for n, item in izip(items_count, items))

    if weight <= bagpack.weight and volume <= bagpack.volume:

        return sum(n * item.value for n, item in izip(items_count, items)), -weight, -volume

    else:

        return -1, 0, 0

def knapsack():

    global items, bagpack

    max1 = [min(int(bagpack.weight // item.weight), int(bagpack.volume // item.volume)) for item in 
items]

    return max(product(*[range(n + 1) for n in max1]), key=tot_value)

max_items = knapsack()

maxvalue, max_weight, max_volume = tot_value(max_items)

max_weight = -max_weight
```

```
max_volume = -max_volume

print("The maximum value achievable (by exhaustive search) is %g." % maxvalue)

item_names = ", ".join(item.name for item in items)

print("  The number of %s items to achieve this is: %s, respectively." % (item_names, max_items))

print("  The weight to carry is %.3g, and the volume used is %.3g." % (max_weight, max_volume))
```

Output-

```
The maximum value achievable (by exhaustive search) is 54500.
  The number of laptop, printer, headphone items to achieve this is: (9, 0, 11), respectively.
  The weight to carry is 24.7, and the volume used is 0.247.
```

Experiment-9

Aim-Preprocessing Techniques in NLP Using NLTK package

Code-