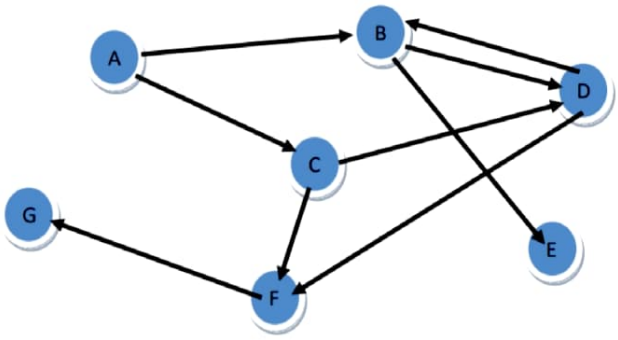
4. End



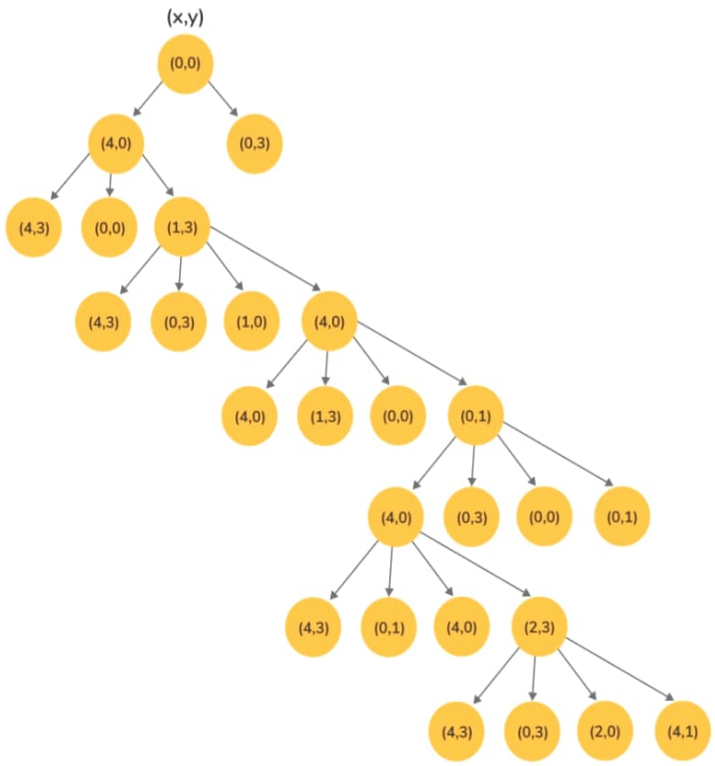

```python
from collections import deque
def bfs(graph, start_node, goal_node):
    # Initialize a queue for BFS
    queue = deque()
    queue.append([start_node])  # Add the start node as the first path
    visited = set()  # To track visited nodes

    while queue:
        # Dequeue the first path
        path = queue.popleft()
        # Get the last node in the path
        current_node = path[-1]

        # Check if we reached the goal node
        if current_node == goal_node:
            return path  # Return the path to the goal node

        # If current node is not visited
        if current_node not in visited:
            visited.add(current_node)

            # Add all unvisited neighbors to the queue
            for neighbor in graph[current_node]:
                new_path = path + [neighbor]
                queue.append(new_path)

    return "Goal not reachable"

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["D", "F"],
    "D": ["B", "F"],
    "E": [],
    "F": ["G"],
    "G": [],
}

# Call the BFS function
start_node = "A"
goal_node = "G"
path_to_goal = bfs(graph, start_node, goal_node)

# Print the result
print("Path to goal node:", path_to_goal)
```

Output: Path to goal node: ['A', 'C', 'F', 'G']

Operations:

1. Fill the 4-liter jug:
   $(x, y) \rightarrow (4, y)$, valid if $x < 4$.
2. Fill the 3-liter jug:
   $(x, y) \rightarrow (x, 3)$, valid if $y < 3$.
3. Empty the 4-liter jug onto the ground:
   $(x, y) \rightarrow (0, y)$, valid if $x > 0$.
4. Empty the 3-liter jug onto the ground:
   $(x, y) \rightarrow (x, 0)$, valid if $y > 0$.
5. Pour water from the 3-liter jug into the 4-liter jug:
   $(x, y) \rightarrow (4, y - (4 - x))$, valid if $0 < x + y \geq 4$ and $y > 0$.
6. Pour water from the 4-liter jug into the 3-liter jug:
   $(x, y) \rightarrow (x - (3 - y), 3)$, valid if $0 < x + y \geq 3$ and $x > 0$.
7. Pour all water from the 3-liter jug into the 4-liter jug:
   $(x, y) \rightarrow (x + y, 0)$, valid if $0 < x + y \leq 4$ and $y \geq 0$.
8. Pour all water from the 4-liter jug into the 3-liter jug:
   $(x, y) \rightarrow (0, x + y)$, valid if $0 < x + y \leq 3$ and $x \geq 0$.

- **Example problem:** X = 4, Y = 3, Target Z = 2 (Jug 1)



```
from collections import deque

def water_jug_bfs(jug1_capacity, jug2_capacity, target):
    # Initialize a queue to store states
    queue = deque()
    # A set to store visited states
    visited = set()

    # Initial state (both jugs empty)
    initial_state = (0, 0)
    queue.append((initial_state, []))  # (state, path leading to this state)
    visited.add(initial_state)
```



```
while queue:
    (current_jug1, current_jug2), path = queue.popleft()

    # If target is achieved
    if current_jug1 == target or current_jug2 == target:
        return path + [(current_jug1, current_jug2)]

    # Generate all possible states
    possible_states = [
        (jug1_capacity, current_jug2),  # Fill Jug1
        (current_jug1, jug2_capacity),  # Fill Jug2
        (0, current_jug2),              # Empty Jug1
        (current_jug1, 0),              # Empty Jug2
        # Pour Jug1 into Jug2
        (max(current_jug1 - (jug2_capacity - current_jug2), 0),
         min(current_jug1 + current_jug2, jug2_capacity)),
        # Pour Jug2 into Jug1
        (min(current_jug1 + current_jug2, jug1_capacity),
         max(current_jug2 - (jug1_capacity - current_jug1), 0))
    ]

    for state in possible_states:
        if state not in visited:
            visited.add(state)
            queue.append((state, path + [(current_jug1, current_jug2)]))

# If no solution exists
return "No solution"
```

```
# Main function to get input from the user
def main():
    print("Welcome to the Water Jug Problem Solver!")

    # Take input from the user for jug capacities and target amount
    jug1_capacity = int(input("Enter the capacity of Jug 1: "))
    jug2_capacity = int(input("Enter the capacity of Jug 2: "))
    target = int(input("Enter the target amount of water: "))

    # Validate the input
    if target > max(jug1_capacity, jug2_capacity):
        print("Target cannot be greater than the capacity of both jugs combined!")
        return

    # Call the BFS function
    result = water_jug_bfs(jug1_capacity, jug2_capacity, target)

    # Display the result
    if result == "No solution":
        print("No solution exists for the given capacities and target.")
    else:
        print("Steps to reach the target:")
        for step in result:
            print(step)

# Run the main function
if __name__ == "__main__":
    main()
```


Output:

```
Welcome to the Water Jug Problem Solver!
Enter the capacity of Jug 1: 4
Enter the capacity of Jug 2: 3
Enter the target amount of water: 2
Steps to reach the target:
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

```
class State:
    def __init__(self, cannibalLeft, missionaryLeft, boat, cannibalRight,
missionaryRight, parent=None):
        self.cL, self.mL, self.boat, self.cR, self.mR = cannibalLeft,
missionaryLeft, boat, cannibalRight, missionaryRight
        self.parent = parent
```



```
    def is_goal(self):
        return self.cR == 0 and self.mR == 0

    def is_valid(self):
        return (0 <= self.mL >= self.cL or self.mL == 0) and \
               (0 <= self.mR >= self.cR or self.mR == 0) and \
               self.cL >= 0 and self.mL >= 0 and self.cR >= 0 and self.mR >= 0

    def __eq__(self, other):
        return (self.cL, self.mL, self.boat, self.cR, self.mR) == \
               (other.cL, other.mL, other.boat, other.cR, other.mR)

    def __hash__(self):
        return hash((self.cL, self.mL, self.boat, self.cR, self.mR))

def successors(state):
    moves = [(2, 0), (0, 2), (1, 1), (1, 0), (0, 1)]  # (cannibals,
missionaries)
    children = []
    for c, m in moves:
        if state.boat == 'right':
            child = State(state.cL + c, state.mL + m, 'left', state.cR - c,
state.mR - m, state)
        else:
            child = State(state.cL - c, state.mL - m, 'right', state.cR + c,
state.mR + m, state)
        if child.is_valid():
            children.append(child)
    return children
```

```
def depth_first_search():
    initial_state = State(0, 0, 'right', 3, 3)
    stack, explored = [initial_state], set()
    while stack:
        state = stack.pop()
        if state.is_goal():
            return state
        explored.add(state)
        for child in successors(state):
            if child not in explored and child not in stack:
                stack.append(child)
    return None
```



```
def print_solution(state):
    path = []
    while state:
        path.append(state)
        state = state.parent
    for step in reversed(path):
        print(f"({step.cL}, {step.mL}, {step.boat}, {step.cR}, {step.mR})")


if __name__ == "__main__":
    solution = depth_first_search()
    print("Missionaries and Cannibals solution using DFS:")
    print("(cannibalLeft, missionaryLeft, boat, cannibalRight,
missionaryRight)")
    if solution:
        print_solution(solution)
    else:
        print("No solution found!")
```

Output:

```
Missionaries and Cannibals solution using DFS:
(cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight)
(0, 0, right, 3, 3)
(1, 1, left, 2, 2)
(1, 0, right, 2, 3)
(3, 0, left, 0, 3)
(2, 0, right, 1, 3)
(2, 2, left, 1, 1)
(1, 1, right, 2, 2)
(1, 3, left, 2, 0)
(0, 3, right, 3, 0)
(2, 3, left, 1, 0)
(2, 2, right, 1, 1)
(3, 3, left, 0, 0)
```

Program:-

```
def towers_of_hanoi(n, source="A", auxiliary="B",
                    destination="C"):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
    towers_of_hanoi(n-1, source, destination, auxiliary)
    print(f"Move disk {n} from {source} to {destination}")
    towers_of_hanoi(n-1, auxiliary, source, destination)
def main():
    n = int(input("Enter the number of disks: "))
    print("\n Steps to solve Towers of Hanoi:")
    towers_of_hanoi(n)
if __name__ == "__main__":
    main()
```

Output:

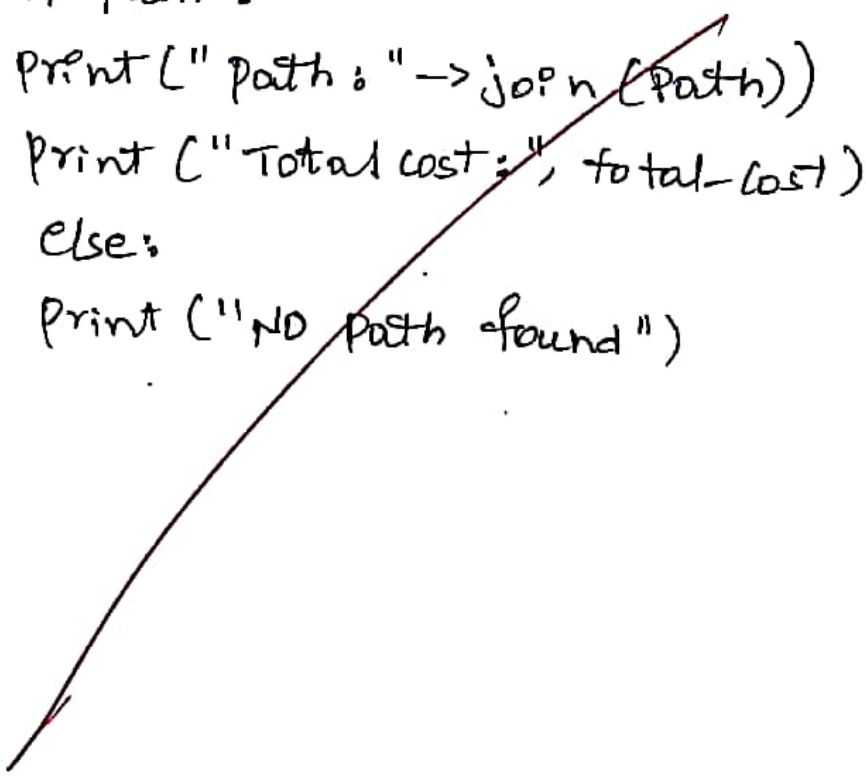
```
Enter the number of disks : 3
Steps to solve Towers of Hanoi:
Move disk 1 from A to C
move disk 2 from A to B
move disk 1 from C to B
move disk 3 from A to C
move disk 1 from B to A
move disk 2 from B to C
move disk 1 from A to C

Enter the number of disks: 4
Steps to solve Towers of Hanoi:
Move disk 1 from A to B
move disk 2 from A to C
move disk 1 from B to C
move disk 3 from A to B
move disk 1 from C to A
move disk 2 from C to B
move disk 1 from A to B
move disk 4 from A to C
move disk 1 from B to C
move disk 2 from B to A
move disk 1 from C to A
```

Aim:- Develop ~~an~~ ~~Knowledge~~ ~~words~~

Source code:-

```
From collection import deque
graph = {
"Kavali" : [("Badvel", 45), ("Nellore", 57)],
"Nellore" : [("Kavali", 57), ("Gudur", 43)],
"Gudur" : [("Srikalasti", 59), ("Naidupeta", 29), ("Nellore", 43)],
"Naidupeta": [("chittor", 160), ("Tada", 16)],
"Tada" : [("Vellore", 190), ("Chennai", 87)],
"Chennai" : [("Tada", 87), ("Arakonnam", 80)],
"Arakonnam": [("Chennai", 35), ("Arakonnam", 78), ("Ambur", 50)],
"Ambur": [("Vellore", 50), ("Krishnagiri", 68), ("Uthagiri", 64)],
"Uthanogari": [("Ambur", 64), ("Salem", 190)],
"Salem": [("Uthagasai", 190), ("Dharmapuri": 69) ("Erode", 67)],
"Erode": [("Salem", 67), ("Thirapur", 53), ("Coaimbattore", 120)],
"Thirapur": [("Comaimbattore", 58)]
}
def bfs(graph, start, goal):
    visited = set()
    queue = deque([start, [start], 0])
    while queue:
        current path, cost = queue.popleft()
        if current == goal:
            return path, cost
    for neighbour, Edge_cost in graph.get(current[]):
        if neighbour not in visited:
```

```
visited.add(neighbour)
queue.append((neighbour, path + [neighbour], cost + edge_cost))
return None,
    o # return None if no path exists.
start_city = "Kavali"
goal_city = "coimbattore"
path, total_cost = bfs(graph, start_city, goal_city)
if path:
print("path:" -> join(path))
print("Total cost:", total_cost)
else:
print("NO path found")
```



Source code:-

```
import heapq
graph = {
"Kakali" = [("Badvel", 115),("Nellore", 57)]
"Nellore" = [("Kakali", 57),("Gudur", 43)]
"Gudur" : [("Srikalahasti", 59),("Naidupeta", 29),("Nellore", 43)]
"Naidupeta" : [("Chittor", 162),("Tada", 6)],
"Tada" : [("vellore", 95),("chennai", 87)]
"chennai" : [("Tada", 87),("Arakonnam", 80)]
"Arakonnam" : [("chennai", 80),("vellore" : 78)]
"vellore" : [("Chittor", 35),("Arakonnam", 78),("Ambur", 50)]
"Ambur" : ("vellore", 50),("Krishnagiri", 68),("uthangiri", 64)]
"uthangiri" : [("Ambur", 69),("salem", 90)]
"salem" : [("uthangiri", 190),("Dharmapuri", 69),("Erode", 67)]
"Erode" : [("salem", 67),("Tirupur", 53),("coimbattore", 120)]
"Tirupur" : [("coimbattore", 52)]
}
heuristic = {
"Kakali": 350, "Badvel": 220", "Gudur": 150, "Naidupeta": 160,
"Tada": 747, "chennai": 96, "vellore": 94, "chittor": 152,
"Krishnagiri": 142, "Dharmapuri": 144, "salem": 151
"Erode": 32, "Nellore": 180, "Srikalahasti": 180,
"Tirupur": 154, "Ambur": 51, "uthagiri" 40, "Tirpur": 174
"Arakonnam": 76 , "coimbattore": 0
```

```
def a-star-search(start, goal):
  open_list = []
heapq.heappush(open_list, (0 + heuristic(start, goal), 0, start,
                                                        start))
    closed_set = set()
  while open_list:
cost, current, path = heapq.heappop(open_list)
  if current == goal:
    return path, cost
  closed_set.add(current)
for neighbour, distance in graph[current]:
  if neighbour not in closed_set:
      g = cost + distance
      f = g + heuristic[neighbour]
heapq.heappush(open_list, (f, g, neighbour, path +
                                        [neighbour]))
      return None, None
source = input("Enter the source city:")
destination = input("Enter the destination city:")
path, cost = a-star_search(source, destination)
  if path:
  path (f" path: {'->'.join(path)} ")
  print(f" Total cost: {cost}")
   else:
  print("No path found.")
```

Program:

```
def count_occurence(s1,s2):
    counter=0
for i in range(len(s2)-len(s1)+1):
    if s2[i:i+len(s1)]==s1:
        counter+=1
    return counter
s2=input("Enter the main string:")
s1=input("Enter the substring to count:")
count=count_occurences(s1,s2)
print(count)
```

output:-

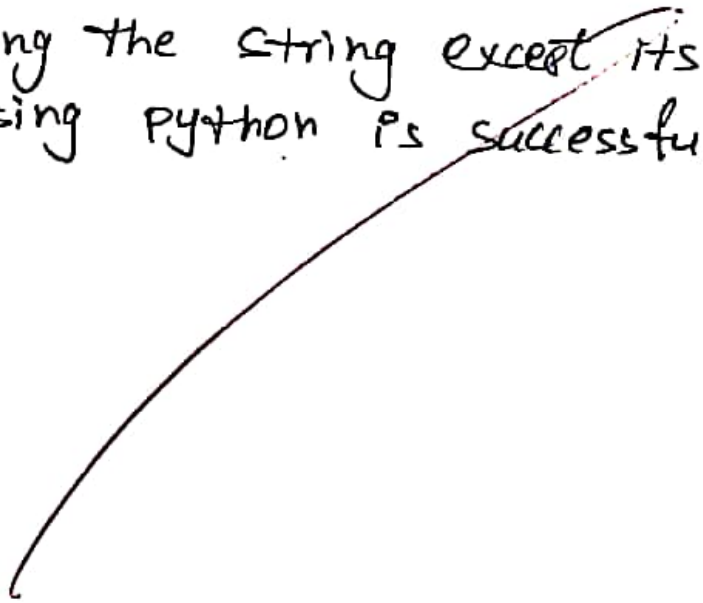
```
Enter the main string :abggaba
Enter the substring to count: aba
2
Enter the main string :
Enter the substring to count :
```

Result: Hence, counting the no. of times a string occurs in another string is successfully completed.

```
def remove-first-char(s)
    return s[1:] if len(s) > 1 else ""
  string = input ("Enter a string:")
print ("string without first character :", remove-
                                    first-char (string))
```

```
Enter a string : python
String without first character: ython
Enter a string :
```

Hence, printing the string except its first character using python is successfully executed.



Program:

```
def count (test , list)
    count = 0
  for item in list :
      if test (item):
          count += 1
        return count
result = count (lamda x: x>0 , [1,2,3,4,5])
print (result)
```

OUTPUT:

3

Result :-

Hence, counting the no. of element in a list that satisfies the given list using higher-order function is succesfully excuted.

Write a Python program to count no. of elements in a list that satisfies the given list without using any existing higher order function

Aim:

To write a Python program to count no. of element in a list that satisfies the given list without using any existing or higher order function

Requirement:

- Knowledge on Python coding
- Python IDE

Program:

```
def count (test, list)
        count = 0
    for item in list:
        if test (item):
            count += 1
        return count
def is_greater_than_two (x)
        return x > 2
result = count (is_greater_than_two [1, 2, 3, 4, 5])
print (result)
```

output:

3

Result:

Hence, counting no. of element in a list without using any existing higher order function is successfully executed.

Date

Roll No. :

Source code:

```
import random
import time
def generate_knapsack_instance(N):
 items=[{"size":random.randint(1,15),"value":random,
        randint(1,10)} for in range(N)]
 capacity=int(2.5*N)
 return items, capacity
def knapsack_capacity_solver(items capacity):
 N= len(items)
 dp=[[0](N(capacity+1) for in range(N+1)]
 for i in range(1,N+1);
 for w in range(capacity+1);
 if ifems[i-1]["size"]<=W:
 dp[i][W]= max[items[i-1]["value"]+dp[i-1]
 [w-items[i-1]["size"]],dp[i-1][w]]
 else:
 dp[i][w]=dp[i-1][w]
  return dp[N][capacity]
Problem-sizes=[10,12,14,16,18,20,22]
 Excuetion-items={}
for N in problem_sizes:
 item, capacity= generate_knapsack_instance(N)
    start_time=time.time()
max-value=knapsack_solver(items, capacity)
```

```
    end_time = time.time()
    Execution_time[N] = end_time - start_time
    print(f"N = {N}, capacity = {capacity}, max value = {max_value}, Execution Time = {Execution_time[N]:.6f} seconds")
print("\n Execution Time Analysis:")
for N, exec_time in Execution_times.items():
print(f"N = {N}: {Execution_time:.6f} seconds")
```

OUTPUT:—

N=10, capacity=25, max value=47,
Execution Time = 0.000024 seconds

N=12, capacity=30, Max value = 48,
Execution Time= 0.000035 seconds

N=14, capacity=35, Max value = 55,
Execution Time= 0.000051 seconds

N=16, capacity=40, Max value = 62,
Execution Time = 0.000071 seconds

N=18, capacity= 45, Max value = 68,
Execution Time = 0.000089 seconds

N=20, capacity = 50, Max value = 75,
Execution Time = 0.000123 seconds.

Source code:

```
from typing import List, Tuple, Dict
def table_assignment(N: int, C: int, L: List
        [Tuple[int, int]]) -> Dict[int, int] or bool:
conflict_graph = {i: set() for i in range(N)}
for x,y in L:
 conflict_graph[x].add[y]
 conflict_graph[y].add[x]
table_assignment = {}
for guest in range(N):
used_tables = {table_assignment[g] for g in
 conflict_graph[guest] if g in table_assignment}
for table in range(C):
   if table not in used_tables:
table_assignment[guest] = table
        break
    else:
    return False
return table_assignment
 N = 5
 C = 3
 L = [(0,1),(1,2),(2,3),(3,4),(4,0)]
 result = table_assignment(N, C, L)
 print("Table Assignment:", result)
```

OUTPUT:

Table Assignment:

{0:0, 1:1, 2:0, 3:1, 4:2}

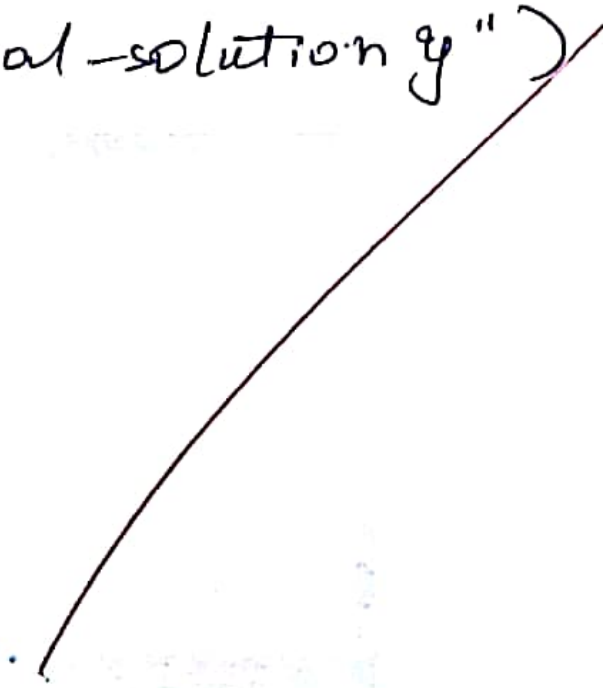
Result:

Hence, the assign N guests to c tables while ensuring that guests with social conflicts do not sit together. If a valid seating arrangement is not possible. The function should return False by using graph and constraints is success fully completed by using python code

Source code:

```
import random
import string
target = "Hello World"
target_length = len(target)
def calculate_score(candidate)
score = sum(1 for a, b in zip(candidate target), if a != b)
return score
 def generate_random_solution():
return join(random.choice(string.asciiletters + ")
 for in range (target_length)
def Mutate(solution):
index = random.randint(0, target_length -1)
new_char = random.choice(string.ascii_letters + " ")
return solution[index] + new_char + solution[index+1:]
 def. Hill_climb_search():
best_solution = generate_random_solution()
best_score = calculate_score(best_solution)
count = 0
while best_score > 0:
candidate_solution = mutate(best_solution)
candidate_score = calculate_score(candidate_solution)
if candidate_score < best_score:
best_solution = candidate_solution
best_score = candidate_score
  count += 1
```

```
print(f" Best score so far: {best_score} cou
      Best_solution: {"best_solution"}")
return best_solution
final_solution = hill_climb_search()
print(f"final solution: {final_solution}")
```



Source Code:

```
import random
import math
def calculate_conflicts(board):
conflicts = 0
 n = len(board)
for i in range(n)
for j in range (i+1, n):
 if board[i] == board[j] .
abs (board[i] - board[j]) == j-i :
  conflicts += 1
return conflicts
def generate_initial_board(n):
return[random.randint(0, n-1) for i in range(n)]
def simulated_annealing(n, initial_temp=1000,
                        cooling_rate=0.95,
                    max_iterations=10000):
board = generate_initial_board(n)
current_temp = initial_temp
 best_board = board
best_cost = calculate_conflicts(board)
for iterations in range(max_iterations):
  if best_cost == 0:
print(f"Solution found at iteration {iteration}");
 return best_board
new_board = board[:]
   i = random.randint(0, n-1)
new_board[r] = random.randint(0, n-1)
```

```
current_cost = calculate_conflicts(board)
new_cost = calculate_conflict(new_board)
if new_cost < current_cost or random.random() < math.exp
current_cost = (new_cost / current_temp);
board = new_board
if new_cost < best_cost:
best_cost = new_cost
best_board = new_board
current_temp *= cooling_rate
print("Solution not found within the iteration limit");
return best_board
def print_board(board):
n = len(board)
for i in range(n):
row = ['Q' if board[i] == j else '.
for j in range(n)
print(" ".join(row))
print("\n")
if __name__ == "__main__":
n = int(input("Enter no. of queens:"))
solution = simulated_annealing(n)
print_board(solution)
```

Source code:-

```
import random
import copy
GOAL_STATE = [[1,2,3],[4,5,6],[7,8,9]]
def Manharran_distance(state):
distance = 0
for i in range(3):
for in range(3):
    value = state[i][i]
      if value != 0;
goal_i, goal_j = divmod(value-1,3)
distance += abs(i-goal_i)+ abs(j-goal_j)
return distance
def generate_initial_state():
numbers = list(range(9))
random.shuffle(numbers)
return [numbers[i:i+3] for i in range(0,9,3)]
def perform_move(state, move):
new_state = copy.deepcopy(state)
zero_i, zero_j = None, None
for i in range(3):
for j in range(3):
if new_state[i][j] == 0:
  zero_i is not None:
  break
if move == 'up' and zero_i > 0;
new_state[zero_i][zero_j] = new_state[zero_i],[zero_j]
```

```
new_state[zero_i][zero_j]
elif move = 'down' and zero_i < 2;
new_state[zero_i][zero_j],
new_state[zero_i+1][zero_j] = new_state[zero_i+1][zero_i]
new_state[zero_i][zero_j]
elif move == 'left' and zero_j > 0;
new_state[zero_i][zero_j]
new_state[zero_i][zero_j-1] = new_state[zero_i][zero_j-1]
newstate[zero_i][zero_j]
return new_state
def hill_climbing(initial_state);
current_state = initial_state
current_heuristic = mahattan_distance(current_state)
iterations = 0
while True:
iteration += 1
next_states = []
for move in ['up', 'down', 'left', 'right']
next_states.append(perform_more current_state, move))
next_heuristic = [manahattan_distance(state) for state
                                                in next_state]
best_heuristic = min(next_heuristic)
if best_heuristic >= current_heuristic:
  break;
best_increase = next_heuristic index(best_heuristic)
current_state = next_states[best_index]
current_heuristic = best_heuristic
if current_state == GOAL_STATE:
  break
return current_state, iterations
```

```
def main ()
initial_state = generate.initial_state()
print ("Initial state:");
for row in initial_state:
  print (row)
final_state, iterations = hill_climbing(initial_state)
print ("final state:");
for row in final_state:
 print (row)
print (f"\n Iterations: {iterations}");
 if __name__ == "__main__":
   main ()
```

OUTPUT:

Intial state:

1	2	3
	4	6
7	5	8

Final state:

1	2	3
4	5	6
7	8	

Result:- 8-puzzle solving using Hill climbing algorithm problem solving using python language is successfully completed.