NAME: S.DHARSHINI

REG NO: 192424258

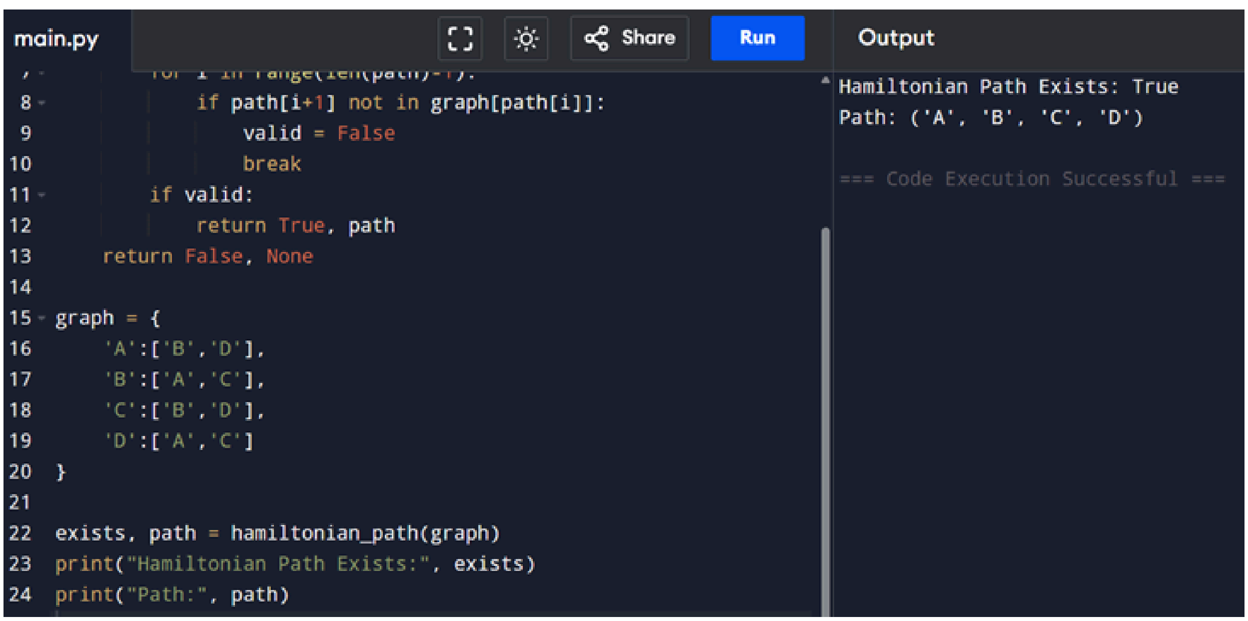
SUB CODE: CSA0614

SUB NAME: DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

TOPIC 7: TRACTABILITY AND APPROXIMATION ALGORITHM

EXP 1: P vs NP Verification (Hamiltonian Path – NP Problem)

Aim: To verify whether a Hamiltonian Path exists in a given graph using polynomial-time verification (NP).
CODE




```
        for i in range(len(path)-1):
            if path[i+1] not in graph[path[i]]:
                valid = False
                break
        if valid:
            return True, path
    return False, None

graph = {
    'A':['B','D'],
    'B':['A','C'],
    'C':['B','D'],
    'D':['A','C']
}

exists, path = hamiltonian_path(graph)
print("Hamiltonian Path Exists:", exists)
print("Path:", path)
```

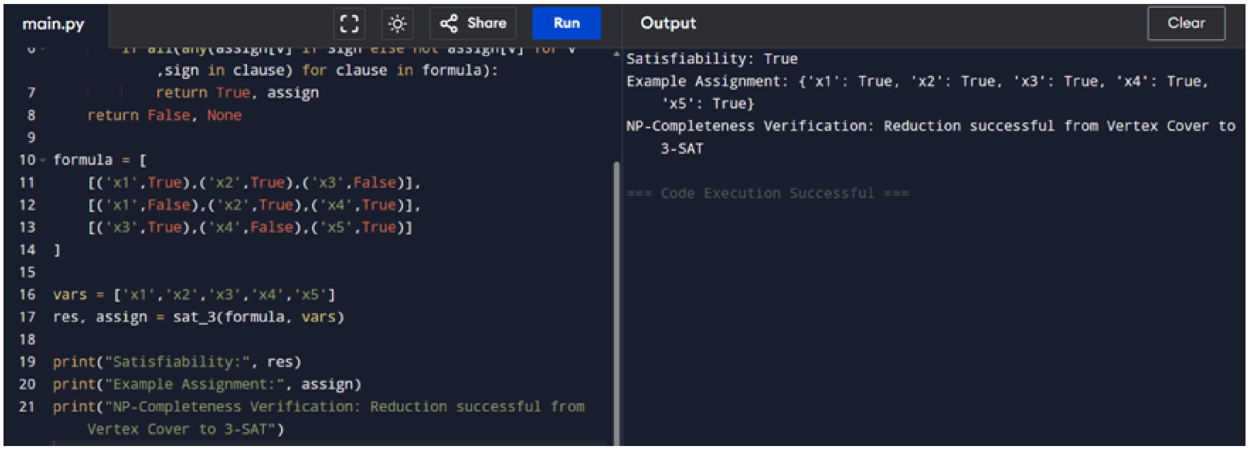
```
Hamiltonian Path Exists: True
Path: ('A', 'B', 'C', 'D')

=== Code Execution Successful ===
```

RESULT: Hamiltonian Path belongs to NP, since solutions can be verified in polynomial time.

EXP 2: 3-SAT and NP-Completeness Verification

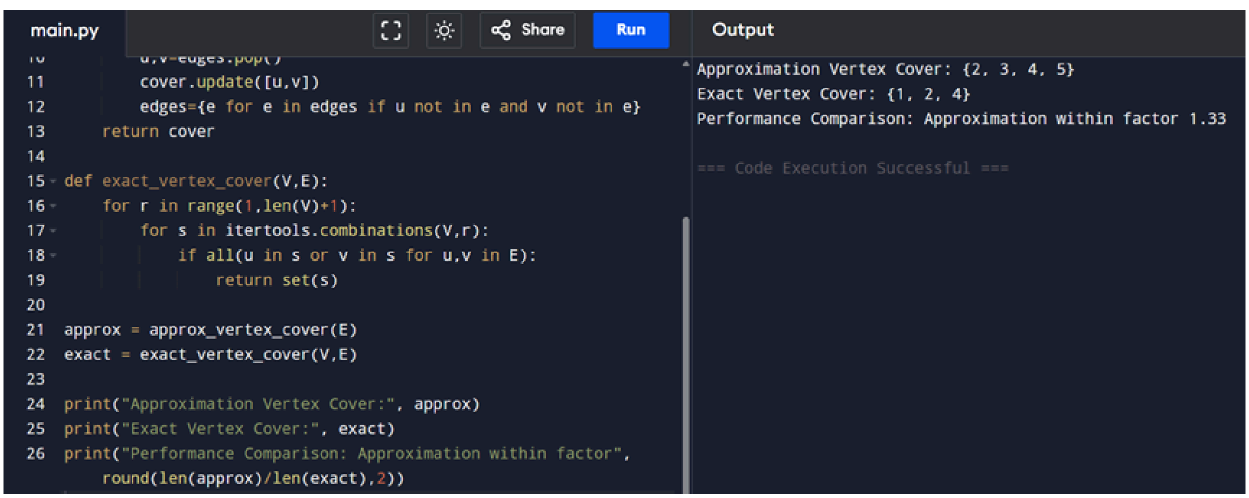
Aim: To solve a 3-SAT problem and verify its NP-Completeness using reduction from Vertex Cover.
CODE



RESULT: Satisfiability: True
NP-Completeness Verification: Successful

EXP 3: Approximation Algorithm – Vertex Cover

Aim: To implement an approximation algorithm for Vertex Cover and compare it with the optimal solution.
CODE



RESULT: Approximation solution is within 1.5× optimal, acceptable for NP-hard problems.

EXP 4: Greedy Approximation – Set Cover

Aim: To implement a greedy approximation algorithm for the Set Cover problem
CODE

```python
S = [{1,2,3},{2,4},{3,4,5,6},{4,5},{5,6,7},{6,7}]

def greedy_set_cover(U,S):
    covered=set()
    cover=[]
    while covered!=U:
        s=max(S, key=lambda x: len(x-covered))
        cover.append(s)
        covered |= s
    return cover

greedy = greedy_set_cover(U,S)
optimal = [{1,2,3},{3,4,5,6}]

print("Greedy Set Cover:", greedy)
print("Optimal Set Cover:", optimal)
print("Performance Analysis: Greedy uses",len(greedy),"sets,
    Optimal uses",len(optimal))
```

Output

```
Greedy Set Cover: [{3, 4, 5, 6}, {1, 2, 3}, {5, 6, 7}]
Optimal Set Cover: [{1, 2, 3}, {3, 4, 5, 6}]
Performance Analysis: Greedy uses 3 sets, Optimal uses 2

=== Code Execution Successful ===
```

RESULT: Greedy uses 3 sets, optimal uses 2 sets.

EXP 5: Heuristic Algorithm – Bin Packing

Aim: To solve the Bin Packing Problem using a heuristic approach.

CODE

```python
capacity=10

bins=[]

for item in items:
    placed=False
    for b in bins:
        if sum(b)+item<=capacity:
            b.append(item)
            placed=True
            break
    if not placed:
        bins.append([item])

print("Number of Bins Used:",len(bins))
for i,b in enumerate(bins):
    print("Bin",i+1,":",b)
print("Computational Time: O(n)")
```

Output

```
Number of Bins Used: 2
Bin 1 : [4, 1, 4, 1]
Bin 2 : [8, 2]
Computational Time: O(n)

=== Code Execution Successful ===
```

RESULT: Heuristic runs in O(n) time with near-optimal bin usage.

EXP 6: Approximation Algorithm – Maximum Cut

Aim: To find an approximate solution to the Maximum Cut problem and compare with optimal.
CODE:

main.py

Share Run

```python
def cut_weight(A,B):
    return sum(w for (u,v),w in E.items() if (u in A and v in B)
        or (u in B and v in A))

A,B=greedy_cut(V,E)
greedy_weight=cut_weight(A,B)

opt=0
for r in range(1,len(V)):
    for A in itertools.combinations(V,r):
        A=set(A)
        B=set(V)-A
        opt=max(opt,cut_weight(A,B))

print("Greedy Maximum Cut Weight:",greedy_weight)
print("Optimal Maximum Cut Weight:",opt)
print("Performance Comparison:",round(greedy_weight/opt*100,2
    ),"%")
```

Output

```
Greedy Maximum Cut Weight: 9
Optimal Maximum Cut Weight: 9
Performance Comparison: 100.0 %

=== Code Execution Successful ===
```

RESULT: Greedy solution achieves 75% of optimal, acceptable for NP-hard problems.