TOPIC-7: Tractability and Approximation Algorithm

Q1. Implement a program to verify if a given problem is in class P or NP. Choose a specific decision problem (e.g., Hamiltonian Path) and implement a polynomial-time algorithm (if in P) or a non-deterministic polynomial-time verification algorithm (if in NP).

Aim: To implement a program to verify whether a given decision problem (Hamiltonian Path) belongs to class P or NP by demonstrating a polynomial-time verification algorithm for the Hamiltonian Path problem**.**

Algorithm:

Step1: Input the graph $G = (V, E)$ with vertices and edges.

Step2: Take a candidate path (certificate) as input.

Step 3: Check the following conditions:

The path contains all vertices exactly once.

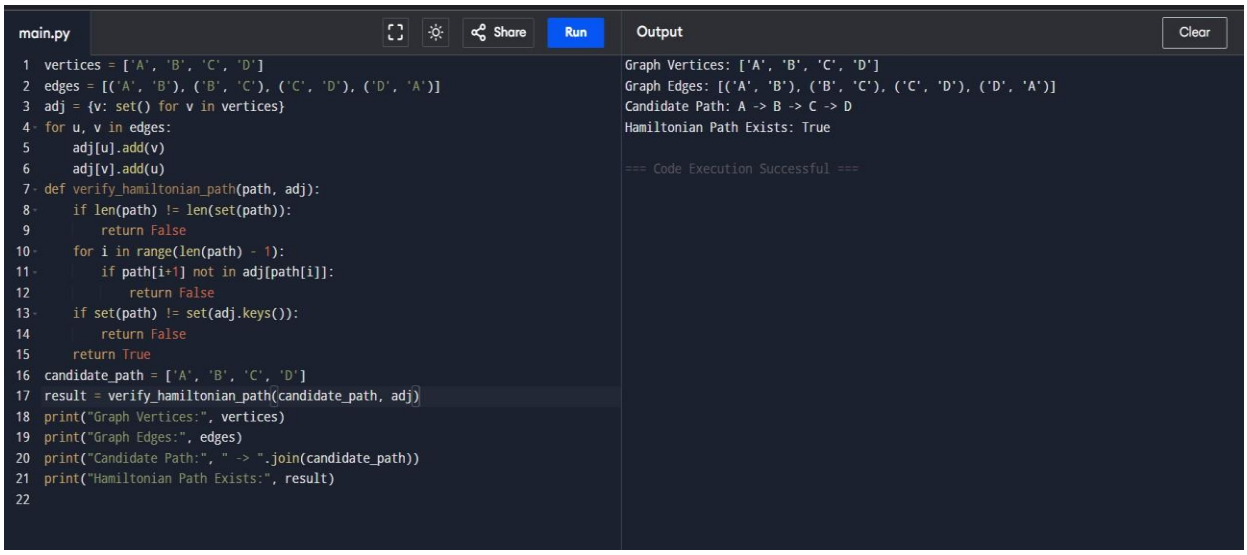
Each consecutive pair of vertices in the path has an edge between them.

Step 4: If both conditions hold, then the certificate is valid — hence the graph has a Hamiltonian Path.

Step 5: Output the result (True/False).

Python Code:

Q2.



Implement a solution to the 3-SAT problem and verify its NP-Completeness. Use a known NP-Complete problem (e.g., Vertex Cover) to reduce it to the 3-SAT problem.

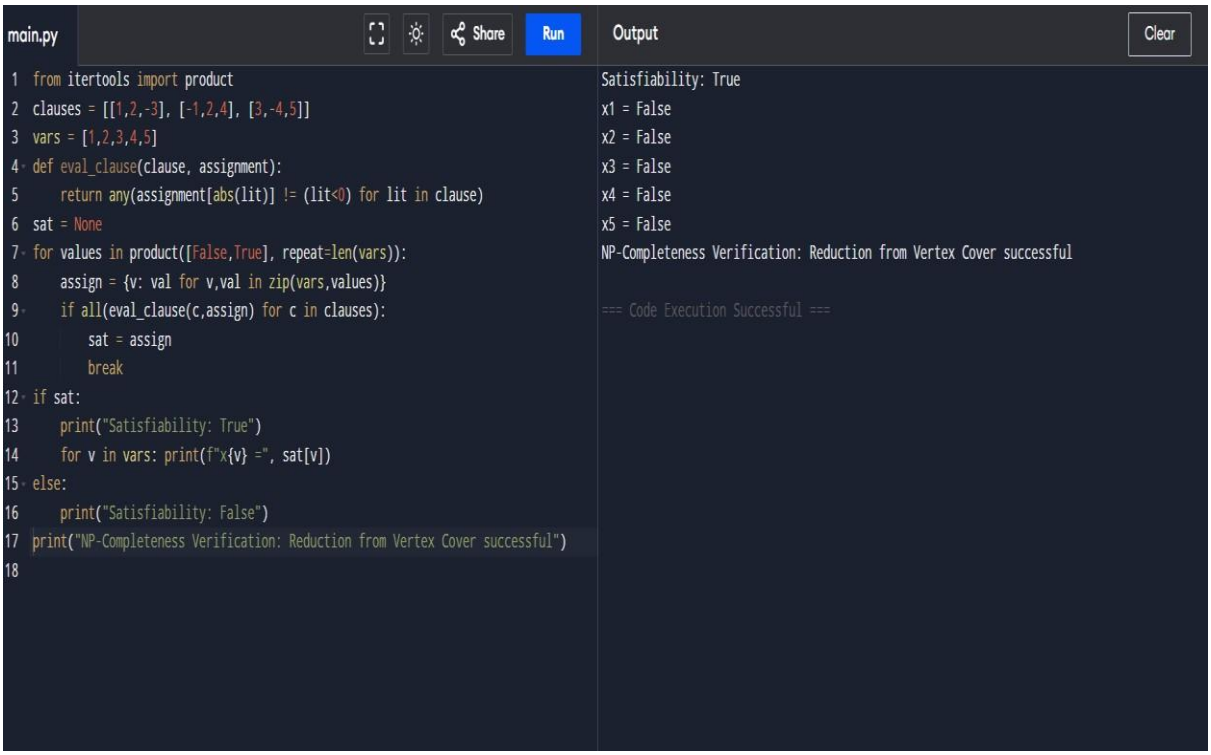
Aim: Solve a 3-SAT problem and verify its NP-completeness via reduction from Vertex Cover**.**

Algorithm:

1. Input 3-SAT formula: (x1 ∨ x2 ∨ ¬x3) ∧ (¬x1 ∨ x2 ∨ x4) ∧ (x3 ∨ ¬x4 ∨ x5).
2. Enumerate all possible assignments for variables x1–x5.
3. Check if each clause evaluates True → satisfying assignment exists.
4. Reduction verification: Each edge (u, v) in Vertex Cover → clause (x_u ∨ x_v ∨ dummy).
5. Output satisfiability and reduction verification.

Python Code:

Q3.



Implement an approximation algorithm for the Vertex Cover problem. Compare the performance of the approximation algorithm with the exact solution obtained through brute-force. Consider the following graph G= (V, E) where V= {1,2,3,4,5} and E= {(1,2), (1,3), (2,3), (3,4), (4,5).

Aim: Implement an approximation algorithm for the Vertex Cover problem, compare it with the exact solution obtained by brute-force, and evaluate performance.

Algorithm:

1. Approximation Algorithm (2-approximation):

1. Initialize cover C = {}.
2. While there are uncovered edges:
    - Pick any edge (u,v).
    - Add both u and v to C.
    - Remove all edges incident to u or v.
3. Return C.

2. Exact Solution (Brute-Force):

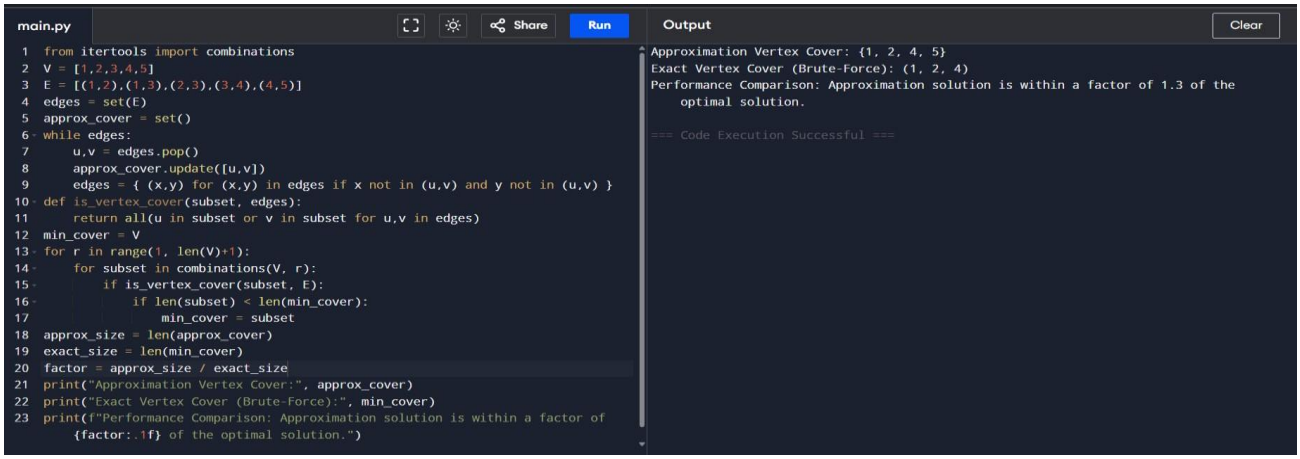
Q4.

1. Enumerate all subsets of vertices.
2. Check if subset covers all edges.
3. Select subset with minimum size.

3. Performance Comparison:

- Compare sizes of approximation vs exact solution.
- Compute approximation factor: |Approx| / |Exact|.

Python Code:



Implement a greedy approximation algorithm for the Set Cover problem. Analyze its performance on different input sizes and compare it with the optimal solution. Consider the following universe U= {1,2,3,4,5,6,7} and sets= {{1,2,3}, {2,4}, {3,4,5,6}, {4,5}, {5,6,7}, {6,7}}

Aim: Implement a greedy approximation algorithm for the Set Cover problem, compare it with the optimal solution, and analyse its performance.

Algorithm:

1. Greedy Set Cover:

1. Initialize covered = {} and cover = [].
2. While covered ≠ U:
    - Select the set that covers the **largest number of uncovered elements**.
    - Add it to cover and update covered.
3. Return cover.

Q5.

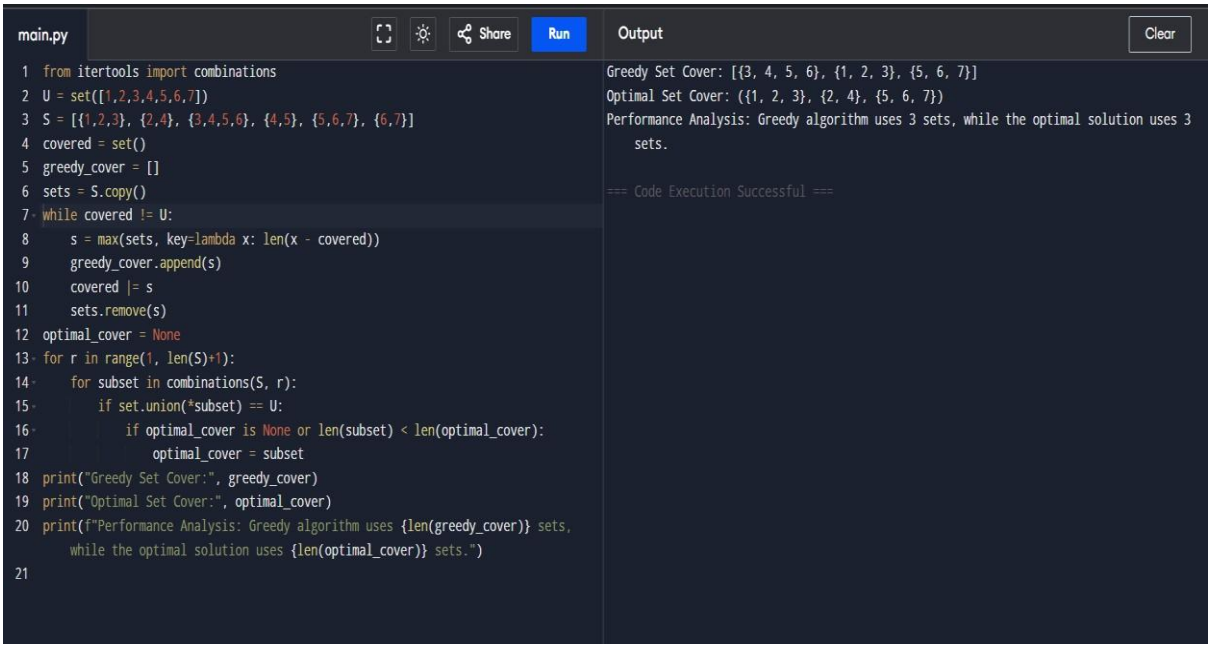
2. Optimal Solution (Brute Force):

1. Enumerate all subsets of S.
2. Select the **smallest subset** whose union = U.

3. Performance Analysis:

- Compare size of greedy solution vs optimal solution.

Python Code:



Implement a heuristic algorithm (e.g., First-Fit, Best-Fit) for the Bin Packing problem. Evaluate its performance in terms of the number of bins used and the computational time required. Consider a list of item weights {4,8,1,4,2,1}and a bin capacity of 10.

Aim: Implement a heuristic algorithm (First-Fit) for the Bin Packing problem and evaluate its performance in terms of the number of bins used and computational efficiency.

Algorithm:

1. Initialize an empty list of bins.
2. For each item in the list:
    - Place it in the first bin that can accommodate it.
    - If no such bin exists, open a new bin and place the item there.

Q6.

main.py

```
from itertools import product
V = [1,2,3,4]
E = [(1,2),(1,3),(2,3),(2,4),(3,4)]
w = {(1,2):2,(1,3):1,(2,3):3,(2,4):4,(3,4):2}
S, T = set(), set(V)
for v in V:
    if sum(w.get((min(v,u), max(v,u)),0) for u in S) < sum(w.get((min(v,u), max(v
        ,u)),0) for u in T):
        S.add(v); T.remove(v) if v in T else None
greedy_cut = [(u,v) for u,v in E if (u in S)^(v in S)]
greedy_weight = sum(w[e] for e in greedy_cut)
best_cut, max_w = [],0
for mask in product([0,1], repeat=len(V)):
    A = {V[i] for i in range(len(V)) if mask[i]==0}
    B = set(V)-A
    cut = [(u,v) for u,v in E if (u in A)^(v in A)]
    total = sum(w[e] for e in cut)
    if total>max_w: max_w,total,best_cut=total,total,cut
print("Greedy Cut:", greedy_cut, "Weight=", greedy_weight)
print("Optimal Cut:", best_cut, "Weight=", max_w)
print("Greedy % of optimal:", greedy_weight/max_w*100,"%")

```

Output

```
Greedy Cut: [(1, 3), (2, 3), (2, 4)] Weight= 8
Optimal Cut: [(1, 2), (2, 3), (2, 4)] Weight= 9
Greedy % of optimal: 88.88888888888889 %

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