

---

**NAME:** S.DHARSHINI

**REG NO:** 192424258

**SUB CODE:** CSA0614

**SUB NAME:** DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

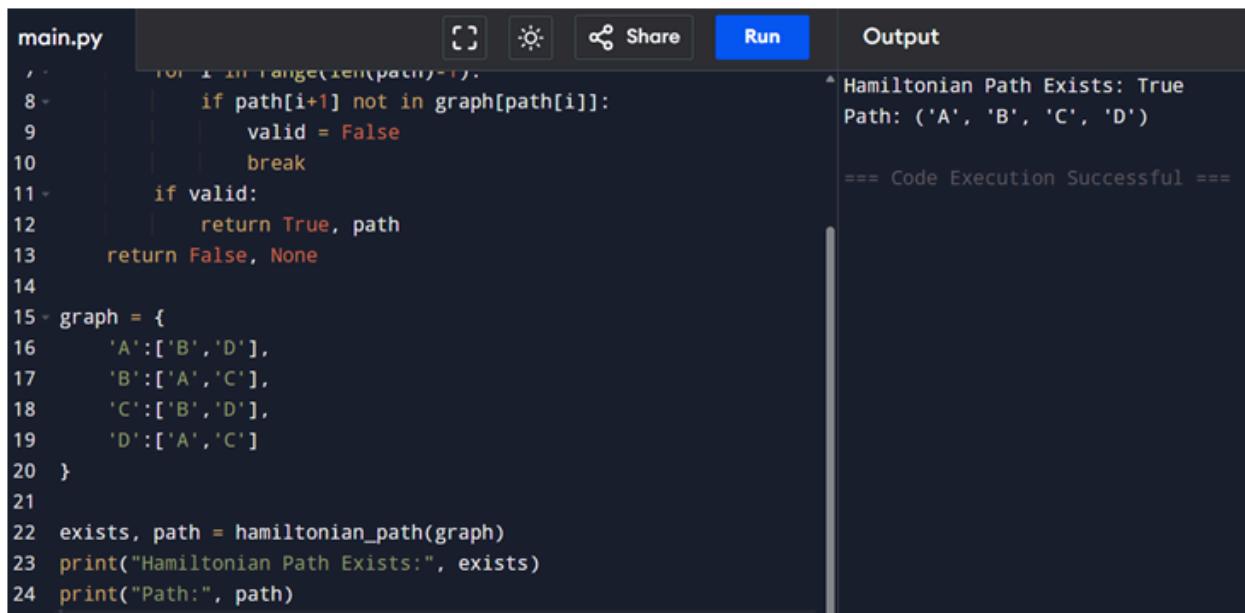
---

## **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**



The screenshot shows a Jupyter Notebook interface with a code cell titled "main.py". The code implements a function to check if a path exists in a graph. It iterates through the path to ensure every vertex has a valid edge to the next vertex. If it finds a vertex where no such edge exists, it breaks the loop and returns False. If it successfully iterates through the entire path, it returns True along with the path itself. The code also defines a sample graph with four nodes (A, B, C, D) and their connections.

```
main.py
1  for i in range(len(path)-1):
2      if path[i+1] not in graph[path[i]]:
3          valid = False
4          break
5      if valid:
6          return True, path
7  return False, None
8
9 graph = {
10     'A': ['B', 'D'],
11     'B': ['A', 'C'],
12     'C': ['B', 'D'],
13     'D': ['A', 'C']
14 }
15
16 exists, path = hamiltonian_path(graph)
17 print("Hamiltonian Path Exists:", exists)
18 print("Path:", path)
```

The "Output" pane shows the results of running the code. It prints "Hamiltonian Path Exists: True" and "Path: ('A', 'B', 'C', 'D')". Below this, it displays "==== 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**

```

main.py
1  def sat_3(formula, vars):
2      if len(formula) == 0:
3          return True, {}
4      if len(formula) == 1:
5          clause = formula[0]
6          if clause[0] == -1:
7              sign_in_clause = False
8          else:
9              sign_in_clause = True
10         for var in vars:
11             if var == clause[1]:
12                 if sign_in_clause:
13                     return True, {var: True}
14                 else:
15                     return False, None
16         return False, None
17
18     formula = [
19         [(-1, 'x1'), (1, 'x2'), (1, 'x3'), (-1, 'x4'), (1, 'x5')],
20         [(-1, 'x1'), (-1, 'x2'), (1, 'x3'), (1, 'x4'), (1, 'x5')],
21         [(1, 'x3'), (-1, 'x4'), (-1, 'x5')]]
22
23     vars = ['x1', 'x2', 'x3', 'x4', 'x5']
24     res, assign = sat_3(formula, vars)
25
26     print("Satisfiability:", res)
27     print("Example Assignment:", assign)
28     print("NP-Completeness Verification: Reduction successful from Vertex Cover to 3-SAT")

```

Satisfiability: True  
Example Assignment: {'x1': True, 'x2': True, 'x3': True, 'x4': True, 'x5': True}  
NP-Completeness Verification: Reduction successful from Vertex Cover to 3-SAT  
== Code Execution Successful ==

**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

```

main.py
1  def approx_vertex_cover(E):
2      cover = set()
3
4      while E:
5          u, v = edges.pop()
6          cover.update([u, v])
7          edges = {e for e in edges if u not in e and v not in e}
8
9      return cover
10
11 def exact_vertex_cover(V, E):
12     for r in range(1, len(V)+1):
13         for s in itertools.combinations(V, r):
14             if all(u in s or v in s for u, v in E):
15                 return set(s)
16
17 approx = approx_vertex_cover(E)
18 exact = exact_vertex_cover(V, E)
19
20 print("Approximation Vertex Cover:", approx)
21 print("Exact Vertex Cover:", exact)
22 print("Performance Comparison: Approximation within factor",
23       round(len(approx)/len(exact), 2))

```

Approximation Vertex Cover: {2, 3, 4, 5}  
Exact Vertex Cover: {1, 2, 4}  
Performance Comparison: Approximation within factor 1.33  
== Code Execution Successful ==

**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

```
main.py | Run | Output
```

```
1 def greedy_set_cover(U,S):
2     covered=set()
3     cover=[]
4     while covered!=U:
5         s=max(S, key=lambda x: len(x-covered))
6         cover.append(s)
7         covered |= s
8     return cover
9
10 greedy = greedy_set_cover({1,2,3},{3,4,5,6})
11 optimal = [{1,2,3},{3,4,5,6}]
12
13 print("Greedy Set Cover:", greedy)
14 print("Optimal Set Cover:", optimal)
15
16 print("Performance Analysis: Greedy uses",len(greedy),"sets,
17       Optimal uses",len(optimal))
```

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**

```
main.py | Run | Output
```

```
1 capacity=10
2
3 bins=[]
4
5 for item in items:
6     placed=False
7     for b in bins:
8         if sum(b)+item<=capacity:
9             b.append(item)
10            placed=True
11            break
12        if not placed:
13            bins.append([item])
14
15
16 print("Number of Bins Used:",len(bins))
17 for i,b in enumerate(bins):
18     print("Bin",i+1,":",b)
19 print("Computational Time: O(n)")
```

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:**

The screenshot shows a Jupyter Notebook cell with the following code in the editor pane:

```
main.py
15 - def cut_weight(A,B):
16     return sum(w for (u,v),w in E.items() if (u in A and v in B)
17               or (u in B and v in A))
18
19 A,B=greedy_cut(V,E)
20 greedy_weight=cut_weight(A,B)
21
22 opt=0
23 for r in range(1,len(V)):
24     for A in itertools.combinations(V,r):
25         A=set(A)
26         B=set(V)-A
27         opt=max(opt,cut_weight(A,B))
28
29 print("Greedy Maximum Cut Weight:",greedy_weight)
30 print("Optimal Maximum Cut Weight:",opt)
31 print("Performance Comparison:",round(greedy_weight/opt*100,2),
32      "%")
```

The output pane shows the results of running the code:

```
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.