NAME : SUMALATHA D K

REG NO: 192424023

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

SLOT : A

TOPIC 7 TRACTABILITY AND APPROXIMATION ALGORITHM

1. **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 verify whether a given decision problem belongs to class **P** or **NP** by choosing a specific problem. Here, we choose the **Hamiltonian Path Problem**, which is an **NP-Complete** problem. We implement a **polynomial-time verification algorithm** to check whether a given path is a valid Hamiltonian path.

ALGORITHM:

1. Take the graph and a proposed path.
2. Check if the path contains all vertices exactly once.
3. Check if each consecutive pair of vertices in the path has an edge.
4. If both conditions are satisfied, return **True**.
5. Otherwise, return **False**.

CODE:

```python
def is_hamiltonian_path(graph, path):
    n = len(graph)
    # Step 1: Check if path contains all vertices exactly once
    if len(path) != n or len(set(path)) != n:
        return False
    # Step 2: Check if consecutive vertices are connected
    for i in range(len(path) - 1):
        if path[i+1] not in graph[path[i]]:
            return False
    return True
# Verification
exists = is_hamiltonian_path(graph, path)
print("Hamiltonian Path Exists:", exists)
if exists:
    print("Path:", " -> ".join(path))
```

INPUT:

```python
# Graph representation (Adjacency List)
graph = {
    'A': ['B', 'D'],
    'B': ['A', 'C'],
    'C': ['B', 'D'],
    'D': ['A', 'C']
}
# Proposed Hamiltonian Path
path = ['A', 'B', 'C', 'D']
```

OUTPUT:

```
Hamiltonian Path Exists: True
Path: A -> B -> C -> D
```

2. 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:

To implement a solution for the **3-SAT problem**, check whether a given Boolean formula is satisfiable, and verify its **NP-Completeness** by demonstrating a reduction from a known NP-Complete problem, **Vertex Cover**, to **3-SAT**.

ALGORITHM:

1. Extract all variables from the formula.
2. Generate all possible truth assignments.
3. Evaluate each clause under the assignment.
4. If all clauses evaluate to True → formula is satisfiable.

5. Return the satisfying assignment.
6. Otherwise, return False.

CODE:

```
import itertools
def evaluate_clause(clause, assignment):
    for literal in clause:
        if literal.startswith('¬'):
            var = literal[1:]
            if not assignment[var]:
                return True
        else:
            if assignment[literal]:
                return True
    return False
def is_satisfiable(clauses, variables):
    for values in itertools.product([True, False], repeat=len(variables)):
        assignment = dict(zip(variables, values))
        if all(evaluate_clause(clause, assignment) for clause in clauses):
            return True, assignment
    return False, None

result, assignment = is_satisfiable(clauses, variables)
print("Satisfiability:", result)
if result:
    print("Satisfying Assignment:")
    for var in assignment:
        print(var, "=", assignment[var])
```

INPUT:

```
# 3-SAT Formula:
clauses = [
    ["x1", "x2", "¬x3"],
    ["¬x1", "x2", "x4"],
    ["x3", "¬x4", "x5"]
]

variables = ["x1", "x2", "x3", "x4", "x5"]
```

OUTPUT:

```
Satisfiability: True
Satisfying Assignment:
x1 = True
x2 = True
x3 = True
x4 = True
x5 = True
```

3. **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:

To implement an **approximation algorithm** for the **Vertex Cover problem** and compare its performance with the **exact solution** obtained using a **brute-force approach**. The goal is to analyze how close the approximation solution is to the optimal solution.

ALGORITHM:

1. Approximation Algorithm (Greedy Method)

Steps:

1. Start with an empty vertex cover set.
2. Pick an arbitrary edge (u, v).
3. Add both u and v to the vertex cover.
4. Remove all edges connected to u and v.
5. Repeat until no edges remain.

2. Exact Algorithm (Brute Force)

Steps:

1. Generate all possible subsets of vertices.
2. For each subset, check if it covers all edges.
3. Return the smallest subset that is a valid vertex cover.

⏱ Time Complexity: **$O(2^n)$** (Exponential)

CODE:

```
import itertools
# Check if a set is a vertex cover
def is_vertex_cover(cover, edges):
    for u, v in edges:
        if u not in cover and v not in cover:
            return False
    return True
# Exact Vertex Cover using brute force
def exact_vertex_cover(vertices, edges):
    n = len(vertices)
    for r in range(1, n + 1):
        for subset in itertools.combinations(vertices, r):
            if is_vertex_cover(set(subset), edges):
                return set(subset)
    return set(vertices)
# Approximation Vertex Cover
def approx_vertex_cover(edges):
    cover = set()
    remaining_edges = edges.copy()
    while remaining_edges:
        u, v = remaining_edges.pop()
        cover.add(u)
        cover.add(v)
        remaining_edges = [e for e in remaining_edges if u not in e and v not in e]
    return cover

approx_cover = approx_vertex_cover(E.copy())
exact_cover = exact_vertex_cover(V, E)
print("Approximation Vertex Cover:", approx_cover)
print("Exact Vertex Cover:", exact_cover)
approx_size = len(approx_cover)
exact_size = len(exact_cover)
ratio = approx_size / exact_size
print("Approximation Ratio:", ratio)
```

INPUT:

```
# Input
V = [1, 2, 3, 4, 5]
E = [(1,2), (1,3), (2,3), (3,4), (4,5)]
```

OUTPUT:

```
Approximation Vertex Cover: {2, 3, 4, 5}
Exact Vertex Cover: {1, 2, 4}
Approximation Ratio: 1.3333333333333333
```

4. **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:

To implement a **greedy approximation algorithm** for the **Set Cover problem**, analyze its performance on a given input, and compare it with the **optimal solution** obtained using brute-force.

ALGORITHM:

1. Greedy Approximation Algorithm

Steps:

1. Initialize covered = ∅
2. Repeat until all elements are covered:
   - Choose the set that covers the **maximum number of uncovered elements**
   - Add it to the solution
   - Update covered elements
3. Return selected sets

Time Complexity: $O(n^2)$

Approximation Ratio: O(log n)

2. Exact Algorithm (Brute Force)

Steps:

1. Generate all possible combinations of sets
2. Check which combinations cover the universe
3. Choose the smallest valid combination

Time Complexity: $O(2^n)$ (Exponential)

CODE:

```
import itertools
def greedy_set_cover(universe, sets):
    covered = set()
    cover = []
    while covered != universe:
        best_set = max(sets, key=lambda s: len(s - covered))
        cover.append(best_set)
        covered |= best_set
    return cover
def exact_set_cover(universe, sets):
    for r in range(1, len(sets) + 1):
        for combo in itertools.combinations(sets, r):
            union = set().union(*combo)
            if union == universe:
                return list(combo)
    return []
greedy_result = greedy_set_cover(U, S)
optimal_result = exact_set_cover(U, S)
print("Greedy Set Cover:", greedy_result)
print("Optimal Set Cover:", optimal_result)
print("Greedy Size:", len(greedy_result))
print("Optimal Size:", len(optimal_result))
```

INPUT:

```
# Input
U = {1, 2, 3, 4, 5, 6, 7}
S = [
    {1, 2, 3},
    {2, 4},
    {3, 4, 5, 6},
    {4, 5},
    {5, 6, 7},
    {6, 7}
]
```

OUTPUT:

```
Greedy Set Cover: [{3, 4, 5, 6}, {1, 2, 3}, {5, 6, 7}]
Optimal Set Cover: [{1, 2, 3}, {2, 4}, {5, 6, 7}]
Greedy Size: 3
Optimal Size: 3
```

5. 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:

To implement a **heuristic algorithm (First-Fit)** for the **Bin Packing Problem** and evaluate its performance based on:

- Number of bins used
- Computational time

ALGORITHM:

Steps:

1. Initialize an empty list of bins.
2. For each item:
    - Try to place it into the **first bin** that has enough remaining capacity.
    - If no such bin exists, create a **new bin**.
3. Repeat until all items are placed.

TIME COMPLEXITY

- Worst-case: **O(n²)**
- Average case: **O(n)** (for small datasets)

CODE:

```
def first_fit_bin_packing(weights, capacity):
    bins = []
    for item in weights:
        placed = False
        for b in bins:
            if sum(b) + item <= capacity:
                b.append(item)
                placed = True
                break
        if not placed:
            bins.append([item])

    return bins

bins = first_fit_bin_packing(weights, capacity)

print("Number of bins used:", len(bins))
for i, b in enumerate(bins):
    print(f"Bin {i+1}:", b)
```

INPUT:

```
# Input
weights = [4, 8, 1, 4, 2, 1]
capacity = 10
```

OUTPUT:

```
Number of bins used: 2
Bin 1: [4, 1, 4, 1]
Bin 2: [8, 2]
```

6. Implement an approximation algorithm for the Maximum Cut problem using a greedy or randomized approach. Compare the results with the optimal solution obtained through an exhaustive search for small graph instances.

AIM:

To implement a **greedy approximation algorithm** for the **Maximum Cut Problem** and compare its result with the **optimal solution** obtained using **exhaustive search** for a small graph.

ALGORITHM:

ALGORITHM 1: GREEDY APPROXIMATION FOR MAX CUT

1. Start with two empty sets A and B.
2. Pick vertices one by one.
3. Place each vertex in the set that maximizes the increase in cut weight.
4. Continue until all vertices are assigned.
5. Compute total cut weight.

ALGORITHM 2: EXHAUSTIVE SEARCH (OPTIMAL)

1. Generate all possible partitions of vertices.
2. For each partition, calculate cut weight.
3. Return the partition with the maximum cut weight.

CODE:

```
# Function to calculate cut weight
def cut_weight(A, B):
    weight = 0
    for (u, v), w in edges.items():
        if (u in A and v in B) or (u in B and v in A):
            weight += w
    return weight
```

```
# Greedy Approach
A, B = set(), set()
for v in vertices:
    gain_A = sum(w for (u, x), w in edges.items() if x == v and u in B or u == v and x in B)
    gain_B = sum(w for (u, x), w in edges.items() if x == v and u in A or u == v and x in A)
    if gain_A > gain_B:
        A.add(v)
    else:
        B.add(v)
greedy_weight = cut_weight(A, B)
# Exhaustive Search
max_weight = 0
best_cut = None
for r in range(len(vertices)):
    for subset in itertools.combinations(vertices, r):
        A = set(subset)
        B = set(vertices) - A
        w = cut_weight(A, B)
        if w > max_weight:
            max_weight = w
            best_cut = (A, B)
print("Greedy Cut Weight:", greedy_weight)
print("Optimal Cut Weight:", max_weight)
```

INPUT:

```
# Graph definition
edges = {
    (1,2): 2,
    (1,3): 1,
    (2,3): 3,
    (2,4): 4,
    (3,4): 2
}
vertices = [1, 2, 3, 4]
```

OUTPUT:

Greedy Cut Weight: 9

Optimal Cut Weight: 9