NP-Completeness & Complexity Theory: A Full Report

Table of Contents

1. Problem Background

Computational complexity theory classifies computational problems based on how efficiently they can be solved. The most prominent complexity classes are:

- **P**: Problems that can be solved in polynomial time by a deterministic Turing machine.
- **NP**: Problems for which a solution can be verified in polynomial time by a deterministic Turing machine.
- **NP-Complete**: The hardest problems in NP. If any NP-complete problem can be solved in polynomial time, then every NP problem can.
- **NP-Hard**: Problems at least as hard as NP-complete problems, but not necessarily verifiable in polynomial time.

Understanding NP-completeness is crucial for solving real-world problems where brute-force solutions are impractical due to their exponential time requirements. This report explores these classes, provides algorithmic examples, dry runs, complexity analysis, and full Python code implementations.

2. Explanation of Key Algorithms

2.1 Prime Number Check (Class P)

Pseudocode

```
function isPrime(n):
    if n <= 1: return False
    for i from 2 to sqrt(n):
        if n mod i == 0:
```

```
            return False
    return True
```

Explanation

This algorithm checks if a number is prime by attempting division from 2 up to its square root. If any divisor is found, the number is not prime.

2.2 Sudoku Validator (Class NP)

Pseudocode

```
function isValidSudoku(board):
    for each row:
        if has duplicates: return False
    for each column:
        if has duplicates: return False
    for 3x3 box:
        if has duplicates: return False
    return True
```

Explanation

This algorithm verifies that a given Sudoku board does not contain duplicates in any row, column, or 3x3 box. A correct solution will satisfy all constraints.

2.3 Vertex Cover (NP-Complete)

Pseudocode

```
function vertex_cover(graph, k):
    for each subset of size k in vertices:
        if all edges are covered:
            return True
    return False
```

Explanation

This brute-force algorithm checks all subsets of vertices of size k to see if they cover all edges in the graph. Although slow, it verifies the existence of a vertex cover.

3. Dry Run or Example

3.1 Prime Check (Input: 11)

- Checks divisibility from 2 to 3.
- No divisors found → 11 is prime.

3.2 Sudoku Validator (3x3 Example)

Input:

```
[
 ['5', '3', '.'],
 ['6', '.', '1'],
 ['.', '9', '8']
]
```

- Rows: Valid
- Columns: Valid
- Box: Valid → Sudoku is valid.

3.3 Vertex Cover (k=2)

Graph: edges = [(1,2), (2,3), (3,4)]

- Try subset (2,3): covers all edges → valid vertex cover

4. Time and Space Complexity Analysis

Algorithm	Time Complexity	Space Complexity
Prime Check	O(sqrt(n))	O(1)
Sudoku Validator	O(n^2)	O(n)
Vertex Cover	O(C(n,k) * m)	O(k)

- **Vertex Cover** is exponential due to combinations C(n,k), where n is the number of vertices and k is the size of the subset.

---

5. Conclusion and Challenges

Understanding NP-completeness provides foundational insight into why certain problems cannot be solved efficiently. Challenges include:

- The P vs NP question remains unsolved.
- No known polynomial algorithms exist for NP-complete problems.
- Real-world applications often require approximations or heuristics.

Despite these challenges, recognizing NP-complete problems allows for better decision-making in algorithm design and resource allocation.

6. References

1. Garey, M.R., & Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*.
2. Sipser, M. (2012). *Introduction to the Theory of Computation*.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*.
4. https://www.geeksforgeeks.org
5. https://leetcode.com

7. Appendix: Full Code

```
# Prime Number Check
import math

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

# Sudoku Validator (Simplified 3x3)
def is_valid_sudoku(board):
    def has_duplicates(values):
        nums = [v for v in values if v != '.']
        return len(nums) != len(set(nums))

    for row in board:
        if has_duplicates(row):
            return False

    for col in range(3):
        column = [board[row][col] for row in range(3)]
        if has_duplicates(column):
            return False

    box = [board[i][j] for i in range(3) for j in range(3)]
    if has_duplicates(box):
        return False

    return True

# Vertex Cover (Brute-force)
```

```python
from itertools import combinations

def is_vertex_cover(graph_edges, vertices, k):
    for subset in combinations(vertices, k):
        cover_set = set(subset)
        if all(u in cover_set or v in cover_set for u, v in graph_edges):
            return True
    return False

# Example Usage
if __name__ == '__main__':
    print("Prime Check (11):", is_prime(11))

    sudoku_board = [
        ['5', '3', '.'],
        ['6', '.', '1'],
        ['.', '9', '8']
    ]
    print("Sudoku Valid:", is_valid_sudoku(sudoku_board))

    edges = [(1,2), (2,3), (3,4)]
    vertices = [1,2,3,4]
    print("Vertex Cover Exists (k=2):", is_vertex_cover(edges, vertices, 2))
```

Group 5 members :

Anas Abdiwahab Mohamed
Shamsa Abdullahi Mohamed
Barwaqo Mohamed Adam
Mohamed Barre Keynan
Mohamed Abdi Mohamed