FINAL PROJECT REPORT

Algorithm Analysis & Design - Final Project

Title: P vs NP Demonstrator
Team Name: ALPHA
Course: Algorithm Analysis & Design

Abstract :

This project presents an experimental study of classical NP and NP-Complete problems in the context of the P vs NP question. We focus on four fundamental problems: **Boolean Satisfiability (SAT), Subset Sum, Vertex Cover, and Hamiltonian Path**. For each problem, both **exact exponential-time algorithms and optimized or approximate algorithms** are implemented and compared. Specifically, we implement the **DPLL algorithm for SAT**, **brute-force and meet-in-the-middle algorithms for Subset Sum**, **exact branching and 2-approximation for Vertex Cover**, and **backtracking and Held-Karp dynamic programming for Hamiltonian Path**.

In addition to solving individual NP problems, the project also demonstrates **polynomial-time reductions**, including a classical reduction from **3-SAT to Vertex Cover** and an illustrative reduction from **Subset Sum to SAT**, thereby experimentally reinforcing the concept of NP-completeness. A complete benchmarking framework was developed to generate random problem instances, measure **runtime and search-tree node exploration**, and verify the correctness of all solutions using automated validation checks.

The experimental results clearly show the **exponential growth of exact algorithms** and the **performance advantages of optimized, dynamic programming, and approximation algorithms**. As a real-world application of SAT, a **4×4 Sudoku puzzle was encoded into CNF and solved using the DPLL solver**. Overall, this project provides both a theoretical and practical understanding of NP-complete problems, reductions, and algorithmic trade-offs, offering strong experimental support to the P vs NP framework.

---

1. INTRODUCTION

The **P vs NP problem** is one of the most fundamental open questions in computer science and mathematics. It asks whether every problem whose solution can be **verified efficiently (in polynomial time)** can also be **solved efficiently**. Problems that can be solved in polynomial time belong to class **P**, while problems whose solutions can only be verified in polynomial time belong to class **NP**. A special subset of NP known as **NP-Complete** contains the hardest problems in NP, such that if any one of them is solved in polynomial time, then all NP problems can be solved in polynomial time.

These problems arise naturally in many real-world applications such as **cryptography, scheduling, circuit design, logistics, artificial intelligence, and network security**. Despite decades of research, no polynomial-time algorithm has yet been discovered for any NP-complete problem, making the P vs NP question one of the **Millennium Prize Problems** with a one million dollar reward for a correct solution.

Instead of attempting to solve the P vs NP problem theoretically, this project focuses on a **practical demonstration of NP and NP-Complete problems**, their **algorithmic behavior**, and their **relationships through reductions**. By implementing multiple algorithms and measuring their performance experimentally, this project helps visualize the challenges posed by exponential complexity.

1.1 PROBLEM STATEMENT

The objective of this project is to **experimentally demonstrate the nature of NP and NP-Complete problems** through implementation, validation, and performance analysis. The project specifically aims to:

- Implement **exact exponential-time algorithms** and compare them with **optimized or approximate algorithms** for key NP-Complete problems.
- Demonstrate **polynomial-time reductions** between NP-Complete problems to verify NP-completeness in practice.
- Collect **runtime and performance data** to observe how computational complexity grows with increasing input size.
- Validate all solutions using **automated correctness checks**.
- Apply SAT solving to a **real-world problem (Sudoku)** to demonstrate practical usefulness.

The project focuses on the following four core NP-Complete problems:

1. **Boolean Satisfiability (SAT)**
2. **Subset Sum**
3. **Vertex Cover**
4. **Hamiltonian Path**

Each problem is solved using more than one algorithm to highlight the trade-offs between **exactness, speed, and scalability**. Additionally, the project includes:

- A **3-SAT to Vertex Cover reduction**
- A **Subset Sum to SAT encoding**
- A **Sudoku to SAT formulation**

Thus, the problem addressed by this project is to **design, implement, benchmark, and analyse multiple NP-Complete problems and their reductions in a unified experimental framework**, providing both theoretical insight and practical evidence regarding computational intractability.

1.2 Objectives

The primary objective of this project is to **experimentally demonstrate the computational difficulty of NP-Complete problems** and to illustrate how different algorithmic strategies affect their performance in practice. Rather than proving P ≠ NP theoretically, this project focuses on a **practical, implementation-based exploration** of NP-Completeness.

The specific objectives of this project are as follows:

1. **To implement exact solvers for multiple NP-Complete problems**, including:
    - SAT
    - Subset Sum
    - Vertex Cover
    - Hamiltonian Path
2. **To implement optimized and heuristic-based algorithms** such as:
    - Meet-in-the-Middle for Subset Sum
    - Held–Karp Dynamic Programming for Hamiltonian Path
    - 2-Approximation algorithm for Vertex Cover
3. **To demonstrate polynomial-time reductions between NP-Complete problems**, specifically:
    - 3-SAT → Vertex Cover
    - Subset Sum → SAT
    - Sudoku → SAT
4. **To empirically measure and analyze time complexity, search space size, and performance growth** using automated benchmarks.
5. **To visualize the exponential behavior of NP-Complete problems** using runtime plots and performance graphs.

6. **To compare exact algorithms with approximations and optimized methods**, highlighting the importance of heuristics in real-world problem solving.
7. **To validate correctness of implementations** using automated assertion and verification checks.
8. **To demonstrate real-world relevance of NP-Complete problems** using Sudoku as a practical SAT application.
9. **To build a complete experimental framework** for algorithm evaluation, benchmarking, and visualization.

1.3 Rationale for Algorithm Selection

In order to effectively demonstrate the practical challenges of NP-Complete problems and the importance of algorithmic design, this project carefully selects **both naive and optimized algorithms** for each problem. The goal is not only to obtain correct solutions, but also to **observe how different algorithmic approaches impact runtime, scalability, and feasibility**.

For the **SAT problem**, the **DPLL algorithm** was selected instead of naive brute-force truth-table checking. While brute-force SAT checking requires evaluating all 2^n possible assignments, DPLL incorporates **unit propagation, pure literal elimination, and intelligent branching**, significantly reducing the search space in many practical instances. This makes DPLL a realistic and industry-relevant SAT solving technique.

For the **Subset Sum problem**, both **brute-force enumeration** and the **Meet-in-the-Middle (MITM)** algorithm were implemented. The brute-force method considers all 2^n subsets and becomes infeasible even for moderate values of n. In contrast, MITM reduces

the complexity to O(2^{n/2}) by splitting the input into two halves. Implementing both allows a direct experimental comparison between naive exponential algorithms and optimized exponential algorithms.

For **Vertex Cover**, two contrasting approaches were chosen:

- An **exact branching algorithm**, which guarantees optimality but has exponential worst-case complexity.
- A **2-approximation greedy algorithm**, which runs in polynomial time and produces near-optimal solutions.

  This comparison highlights the importance of **approximation algorithms** in settings where exact solutions become computationally infeasible.

For the **Hamiltonian Path problem**, two methods were implemented:

- A **backtracking depth-first search**, which explores all possible vertex permutations.
- The **Held–Karp dynamic programming algorithm**, which uses bit masking and achieves significantly better performance for small and medium-sized graphs.

  This demonstrates the role of **dynamic programming in improving exponential algorithms**.

Finally, **SAT was chosen as the central universal problem** for reductions because of its fundamental role in NP-Completeness theory. Reductions from **3-SAT to Vertex Cover**, **Subset Sum to**

SAT, and Sudoku to SAT were implemented to demonstrate how diverse problems can be transformed into logic satisfiability.

Thus, the algorithms in this project were selected not only for correctness, but also to systematically demonstrate the impact of brute force, optimization, approximation, dynamic programming, and problem reductions on NP-Complete problems.

---

2. THEORETICAL ANALYSIS

This section presents the theoretical foundations of the algorithms used to solve the selected NP-Complete problems. For each algorithm, the core idea, working principle, time complexity, strengths, and weaknesses are discussed.

2.1 DPLL Algorithm for SAT

Core Idea

The DPLL (Davis–Putnam–Logemann–Loveman) algorithm is a backtracking-based complete search algorithm used to solve the Boolean Satisfiability (SAT) problem. It improves naive brute-force search using logical simplifications and pruning.

Algorithm Description

The algorithm works recursively by:

1. Applying unit clause propagation
2. Eliminating pure literals
3. Selecting an unassigned variable and branching
4. Recursively exploring both truth assignments
5. Backtracking when conflicts occur

This continues until either a satisfying assignment is found or all possibilities are exhausted.

Time Complexity

- **Worst-case:** O(2^n)

Space Complexity

- O(n), due to recursion and variable assignment storage

Strengths

- Much faster than brute-force SAT
- Uses powerful pruning strategies
- Widely used in real-world SAT solvers

Weaknesses

- Still exponential in the worst case
- Performance heavily depends on variable selection heuristics

2.2 Subset Sum Algorithms

2.2.1 Brute Force Subset Sum

Core Idea

The brute force approach to the Subset Sum problem is based on the observation that any subset of a given set may potentially sum to the target value. Therefore, all possible subsets of the input set are exhaustively generated and tested. If any subset's sum equals the target, the problem is declared solvable.

Algorithm Description

1. Generate all possible subsets of the input set using bit masking.
2. For each subset:
    - Compute the sum of the selected elements.
3. If the computed sum equals the target value, return the subset as a valid solution.
4. If all subsets are exhausted without finding a match, return that no solution exists.

Time Complexity

The algorithm generates all possible subsets of a set of size n, which is 2^n. Therefore, the worst-case time complexity is: O(2^n)

Space Complexity

The algorithm only stores one subset at a time along with recursion or iteration overhead. Hence, the space complexity is: O(n)

Strengths

- Very **simple and easy to implement**.
- Guarantees **correctness and optimality**.
- Works efficiently for **very small input sizes**.

Weaknesses

- **Not scalable** for large input sizes.
- Becomes computationally infeasible even for moderate values of n due to exponential growth.
- Impractical for real-world datasets.

2.2.2 Meet-in-the-Middle (MITM) Subset Sum

Core Idea

The Meet-in-the-Middle (MITM) strategy reduces the exponential search space by **dividing the input set into two equal halves**. Instead of considering all 2^n subsets at once, each half requires only 2^{n/2} subset combinations. The two halves are then efficiently combined to determine whether a subset summing to the target exists.

Algorithm Description

The algorithm operates as follows:

1. Split the input set of size n into two halves of size n/2.
2. Generate and store **all subset sums of the first half**.
3. Generate and store **all subset sums of the second half**.

4. Sort one list of subset sums.
5. For each sum in the first list:
    - Use **binary search** on the second list to find whether a complementary sum exists such that their total equals the target value.
6. If a valid pair of subset sums is found, reconstruct and return the corresponding subset. Otherwise, report that no solution exists.

Time Complexity

Each half produces 2^(n/2) subsets. Sorting and searching introduce a logarithmic factor. Therefore, the total time complexity is: $O(2^{n/2} \log(2^{n/2}))$

Space Complexity

The algorithm stores all subset sums of one or both halves in memory. Hence, the space complexity is:
$O(2^{n/2})$

Strengths

- **Exponentially faster** than the brute-force approach.
- Capable of handling **moderate input sizes** that are completely infeasible using brute force.
- Demonstrates the power of **divide-and-conquer strategies** in exponential problems.

Weaknesses

- Requires large memory to store intermediate subset sums.
- Still has exponential time and space complexity.
- Not feasible for very large values of n.

2.3 Vertex Cover Algorithms

The **Vertex Cover problem** asks whether a given graph contains a set of at most k vertices such that every edge in the graph is incident to at least one selected vertex. This problem is NP-Complete. In this project, both an **exact exponential-time algorithm** and a **polynomial-time approximation algorithm** are implemented to compare optimality and efficiency.

2.3.1 Exact Branching Algorithm

Core Idea

For any uncovered edge in the graph, at least one of its two endpoints must belong to the vertex cover. The algorithm uses this property to systematically branch on each uncovered edge and explore all valid possibilities.

Algorithm Description

The algorithm proceeds as follows:

1. Select an arbitrary uncovered edge (u, v).
2. Create two recursive branches:
    - In the first branch, include vertex u in the cover.
    - In the second branch, include vertex v in the cover.

3. After including a vertex, all its incident edges are removed.
4. The recursion continues until either:
    - All edges are covered (solution found), or
    - The size limit k is exceeded (branch is terminated).

Time Complexity

The algorithm explores two branches for each selected edge. Therefore, the worst-case time complexity is: O(2^k)

where k is the size of the vertex cover.

Space Complexity

The algorithm stores the graph and recursion stack, resulting in: O(V + E)

Strengths

- **Guarantees an optimal solution**.
- Produces a **minimum vertex cover** when a solution exists.
- Useful for small graphs or small values of k.

Weaknesses

- Becomes computationally infeasible for large graphs.
- Exponential growth makes it unsuitable for large-scale applications.

2.3.2 2-Approximation Algorithm for Vertex Cover

Core Idea

Instead of searching for the optimal solution, this greedy heuristic selects **both endpoints of an uncovered edge**, ensuring that the chosen set of vertices forms a vertex cover of size at most **twice the optimal**.

Algorithm Description

The approximate algorithm works as follows:

1. Select any uncovered edge (u, v).
2. Add both vertices u and v to the vertex cover.
3. Remove all edges incident to either u or v.
4. Repeat the process until no uncovered edges remain.

Time Complexity

Each edge is processed at most once. Therefore, the time complexity is: O(E)

Space Complexity

The space complexity is:O(V + E)

Strengths

- Very fast and scalable.
- Suitable for large graphs.
- Provides a guaranteed approximation ratio of 2.

Weaknesses

- Does not guarantee an optimal solution.
- The resulting vertex cover may be larger than the minimum possible one.

2.4 Hamiltonian Path Algorithms

The **Hamiltonian Path problem** determines whether a given graph contains a path that visits **each vertex exactly once**. This problem is NP-Complete and is widely used to model routing, scheduling, and sequencing problems. In this project, both a **brute-force backtracking algorithm** and an **optimized dynamic programming algorithm (Held–Karp)** are implemented to compare naive and optimized exponential approaches.

2.4.1 Backtracking Algorithm

Core Idea

The backtracking approach attempts to generate all possible permutations of the vertices and checks whether any ordering forms a valid Hamiltonian path. It systematically explores all possibilities and backtracks whenever a partial path cannot be extended further.

Algorithm Description

The algorithm proceeds as follows:

1. Start a **depth-first search (DFS)** from each vertex in the graph.
2. Mark the starting vertex as visited.
3. Recursively extend the current path by visiting unvisited neighbouring vertices.
4. If a vertex has no unvisited neighbours, **backtrack** to the previous state.
5. If a path includes all vertices exactly once, a Hamiltonian path is found.

Time Complexity

The algorithm potentially explores all permutations of n vertices. Therefore, the worst-case time complexity is:O(n!)

Space Complexity

The algorithm stores only the current recursion stack and the visited array. Hence, the space complexity is:O(n)

Strengths

- **Very simple and intuitive** to implement.
- **Guarantees correctness** by exploring all possibilities.
- Useful for **small graphs**.

Weaknesses

- **Completely infeasible for large graphs**.
- Factorial time complexity leads to extremely rapid growth in runtime.

2.4.2 Held–Karp Algorithm (Dynamic Programming)

Core Idea

The Held–Karp algorithm improves over brute-force backtracking by using **dynamic programming with bit masking**. Instead of recomputing the same sub paths repeatedly, it stores intermediate results for **subsets of vertices**, significantly reducing redundant work.

Algorithm Description

The algorithm operates as follows:

1. Represent each subset of vertices using a bit mask.
2. Maintain a DP table where each entry stores whether a Hamiltonian path exists for a given subset ending at a particular vertex.
3. Begin with all single-vertex subsets as base cases.
4. Iteratively extend smaller valid paths into larger subsets by adding one vertex at a time.
5. After filling the DP table, reconstruct the Hamiltonian path using the stored predecessor information if a full-length path exists.

Time Complexity

The DP table has 2^n subsets and transitions for n vertices. Therefore, the time complexity is: O(n^2 .2^n)

Space Complexity

The DP table stores values for all subsets and vertices. Hence, the space complexity is: O(n.2^n)

Strengths

- **Significantly faster than naive backtracking** for small and medium-sized graphs.
- **Avoids redundant recomputation** using memoization.
- Provides a practical optimization over factorial-time algorithms.

Weaknesses

- **Very high memory consumption** due to the DP table.
- Still has **exponential time and space complexity**, limiting scalability.

2.5 Real-World SAT Application: Sudoku

Sudoku is a well-known constraint satisfaction problem that can be naturally modelled as a Boolean Satisfiability (SAT) problem. Each valid Sudoku solution corresponds to a satisfying assignment of a carefully constructed Boolean formula. In this project, a **4×4 Sudoku puzzle is encoded into CNF form and solved using the DPLL SAT solver**, demonstrating the real-world applicability of NP-Complete problem solving

Core Idea

Each constraint of the Sudoku puzzle—such as uniqueness in rows, columns, and subgrids—is represented as a **Boolean formula in Conjunctive Normal Form (CNF)**. Solving the Sudoku then reduces to finding a satisfying assignment for this CNF formula using a SAT solver.

Algorithm Description

1. Each Boolean variable represents whether a specific **digit is placed in a specific cell**.
2. **Cell constraints** ensure that each cell contains exactly one digit.
3. **Row constraints** ensure that each digit appears exactly once in every row.
4. **Column constraints** ensure that each digit appears exactly once in every column.
5. **Subgrid constraints** enforce uniqueness within each 2×2 block (for the 4×4 puzzle).
6. The entire constraint set is converted into **CNF format**.
7. The resulting Boolean formula is solved using the **DPLL SAT algorithm**.
8. The satisfying assignment is decoded back into a valid Sudoku grid.

Time Complexity

The time complexity of SAT-based Sudoku solving is:

Exponential in the number of variables

since it ultimately depends on the complexity of the underlying SAT solver.

Strengths

- Demonstrates a **practical real-world application of SAT**.
- Clearly shows how **constraint satisfaction problems can be transformed into Boolean logic**.
- Provides a strong link between **theoretical NP-Completeness and real applications**.

Weaknesses

- Becomes **computationally expensive for larger puzzles**, such as the standard 9×9 Sudoku.
- Requires **advanced SAT solvers and heuristics** for efficient large-scale solving.

3. IMPLEMENTATION DETAILS

This section describes the internal design and implementation of the algorithms used in this project. The implementation focuses on modularity, correctness, experimental evaluation, and automated validation. All algorithms were implemented in **Python**, and the project was organized into multiple solver, reduction, benchmarking, and utility modules.

3.1 Graph Representation

Several algorithms in this project operate on graphs, including **Vertex Cover** and **Hamiltonian Path**. All graphs are represented using an **adjacency list** format for efficiency.

Input Format

- Graphs are represented as a dictionary:
  - Key → Vertex
  - Value → Set of adjacent vertices
- Vertices are labelled using integers.
- Edges are undirected.

Example input:

```
{
  1: {2, 3},
  2: {1, 4},
  3: {1},
  4: {2}
}
```

Output Format

- For Vertex Cover → List of selected vertices
- For Hamiltonian Path → Ordered list of vertices forming the path

This representation provides:

- Fast edge lookup
- Easy edge removal
- Efficient traversal for DFS and branching algorithms

3.2 SAT Solver (DPLL) Implementation

The SAT solver is implemented using the DPLL backtracking algorithm.

Key Features

- Recursive depth-first backtracking
- Unit clause propagation
- Pure literal elimination
- Intelligent variable selection
- Tracks number of search nodes explored

Input Format

- Number of variables: n
- Clauses in CNF format, represented as a list of lists of integers: [[1, -3, 4], [-2, 3], [1, -4]]

Output Format

- True/False indicating satisfiability
- A model (truth assignment)
- Number of nodes explored

3.3 Subset Sum Implementation

Two algorithms are implemented:

(a) Brute Force

- Uses **bit masking** to generate all possible subsets.
- Computes sum for each subset.
- Stops when a valid subset is found.
- Tracks number of subsets explored.

(b) Meet-in-the-Middle (MITM)

- Splits input into two halves.
- Computes all partial subset sums.
- Uses binary search to find complementary sums.
- Extremely faster than brute force for moderate sizes.

Input Format

- List of integers: nums

- Target sum: targe

Output Format

- True/False
- Subset forming the target sum
- Number of explored combinations

3.4 Vertex Cover Implementation

Two approaches are implemented:

(a) Exact Branching Algorithm

- Recursively selects an uncovered edge
- Branches on both endpoints
- Reduces graph size each step
- Guarantees optimal solution

(b) 2-Approximation Algorithm

- Greedily selects both endpoints of uncovered edges
- Removes incident edges
- Produces a cover of size $\leq 2 \times$ optimal

Input Format

- Graph adjacency list
- Maximum cover size k (for exact)

Output Format

- {True/False} whether solution exists
- List of vertices in the cover
- Nodes explored (exact)

3.5 Hamiltonian Path Implementation

Two algorithms are used:

(a) Backtracking DFS

- Starts DFS from every vertex
- Marks visited nodes
- Backtracks on dead ends
- Terminates when full path is found

(b) Held–Karp Dynamic Programming

- Uses bit masks to represent subsets
- DP table stores reachable states
- Reconstructs path via predecessors

Input Format

- Graph adjacency list

Output Format

- {True/False} if Hamiltonian path exists
- Ordered list of vertices in the path
- Node/transition count

3.6 Polynomial-Time Reduction Implementation

The following reductions were implemented:

(a) 3-SAT → Vertex Cover

- Each variable becomes a pair of vertices
- Each clause becomes a triangle
- Edges enforce consistency
- Exact VC solver confirms satisfiability

(b) Subset Sum → SAT

- Each number mapped to Boolean selection bits
- Bitwise parity constraints enforce sum
- Demonstrates logical encoding limitations

3.7 Sudoku → SAT Encoding

- Each cell-digit assignment is a Boolean variable
- CNF constraints enforce:
  - Exactly one digit per cell
  - Unique digits per row, column, and sub grid
- Solved using internal DPLL solver

3.8 Validation & Correctness Checking

A full validation framework is implemented using:

- check_sat
- check_vertex_cover
- check_subset_sum
- check_hamiltonian_path

Additionally:

- A **reduction validation module** tests SAT ↔ Vertex Cover equivalence.
- A **parity mismatch search** exposes logical limits in the SubsetSum → SAT encoding.

4. EXPERIMENTAL SETUP

This section describes how the experimental datasets were generated, how benchmarking was performed, and how the collected performance data was analysed. The goal of the experimental setup is to empirically evaluate the growth in computational cost of NP-Complete problems, compare exact and optimized algorithms, and validate the correctness of polynomial-time reductions.

4.1 Dataset Generation

The datasets were generated programmatically using random instance generators designed separately for each NP-Complete problem. The goal was to **systematically vary the problem size and structure** to observe its effect on algorithm performance.

4.1.1 SAT Instance Generation

Random 3-CNF SAT formulas were generated with:

- Variable count n ranging from 10 to 22
- Clause count fixed at 4n

Each clause contained exactly three distinct literals, selected randomly with random polarity. This ensured:

- Uniform randomness
- Controlled exponential growth
- Compatibility with the 3-SAT → Vertex Cover reduction

4.1.2 Subset Sum Instance Generation

For Subset Sum:

- Input set size n ranged from **10 to 36**
- Each number was sampled uniformly from **[1, 1000]**
- The target sum was chosen as the **sum of a subset of the input numbers**

This ensured that:

- Valid solutions often existed
- Both **Brute Force and MITM** could be compared fairly
- Performance growth could be observed clearly

4.1.3 Vertex Cover Graph Generation

Random undirected graphs were generated with:

- Vertex count n in the range **6 to 14**
- Edge count approximately **2n**

Edges were chosen randomly while avoiding self-loops and duplicates.
The cover size parameter k was set to:

k = [k/2] floor function

This allowed evaluation of:

- Exponential branching behavior
- Approximation quality vs optimal solutions

4.1.4 Hamiltonian Path Graph Generation

Hamiltonian Path instances were generated using:

- Random graphs with vertex count n from 6 to 14
- Edge probability $p \approx 0.4$

This density produces both:

- Graphs that contain Hamiltonian paths
- Graphs that do not contain Hamiltonian paths

ensuring balanced performance evaluation.

4.1.5 Sudoku Test Dataset

A predefined 4×4 Sudoku puzzle was used as the real-world SAT application. The puzzle includes:

- Partially filled cells
- A guaranteed unique solution

This allowed validation of:

- SAT encoding correctness
- Practical use of DPLL

4.2 Benchmarking Protocol

A fully automated benchmarking framework was developed using Python. Each algorithm was executed repeatedly on increasing problem sizes while measuring:

- **Wall-clock runtime**
- **Search-tree nodes explored**
- **Number of states evaluated**
- **Validation success/failure**

4.2.1 Execution Method

Each benchmark run follows this sequence:

1. A random instance is generated.
2. The corresponding solver is executed.
3. Execution time is measured using time.time().
4. Output is automatically validated using:
    - check_sat
    - check_subset_sum
    - check_vertex_cover
    - check_hamiltonian_path
5. Results are written to CSV files.

4.2.2 Benchmarked Algorithms

Problem	Algorithms Benchmarked
SAT	DPLL
Subset Sum	Brute Force, Meet-in-the-Middle
Vertex Cover	Exact Branching, 2-Approximation

Problem	Algorithms Benchmarked
Hamiltonian Path	Backtracking, Held–Karp DP

4.2.3 Output Data Files

All benchmarking results are stored in structured CSV files:

- sat_runtime.csv
- subsetsum_runtime.csv
- vertexcover_runtime.csv
- hampath_runtime.csv

Each row contains:

- Input size
- Algorithm name
- Runtime (seconds)
- Nodes explored
- Whether a solution was found

4.3 Analysis Pipeline

After completing all benchmark executions, the collected performance data was processed through a **Python-based analysis and visualization pipeline**. This pipeline aggregates runtime data, evaluates solver efficiency, verifies correctness, and produces visual insights into algorithm behavior across increasing problem sizes.

The analysis focuses on understanding:

- Growth patterns of exponential algorithms
- Impact of optimization techniques
- Practical feasibility of exact solvers
- Effectiveness of approximation algorithms

4.3.1 Statistical Analysis

For each algorithm and each problem instance, the following statistical metrics were computed:

- **Mean Runtime:** Average execution time across all tested instances
- **Maximum Runtime:** Worst-case observed execution time
- **Search-Tree Growth:** Number of explored nodes or transitions (for SAT, Vertex Cover, and Hamiltonian Path)
- **Success Rate:** Percentage of instances in which the algorithm successfully found a valid solution

These statistics allow a quantitative comparison between:

- Exact vs optimized algorithms
- Brute-force vs heuristic-based approaches
- Polynomial-time vs exponential-time procedures

4.3.2 Visualization

To clearly illustrate performance trends, a dedicated plotting module was implemented using **Matplotlib**. The following visualizations were generated:

- **SAT Runtime vs Number of Variables**
- **Subset Sum:** Brute Force vs Meet-in-the-Middle (MITM)
- **Vertex Cover:** Exact Branching vs 2-Approximation Algorithm

- **Hamiltonian Path:** Backtracking vs Held–Karp Dynamic Programming

These plots clearly demonstrate:

- The **exponential growth** of NP-Complete problems
- The **performance improvements obtained by intelligent optimizations**
- The **scalability limits of exact algorithms** even for moderately sized inputs
- The **efficiency of approximation algorithms** for large problem instances

All plots are automatically saved as PNG files for inclusion in the project report.

4.3.3 Reduction Validation Analysis

In addition to performance analysis, the **correctness of polynomial-time reductions** was empirically validated using a dedicated validation module.

The following validations were performed:

- **3-SAT ⇔ Vertex Cover Reduction:**

  Random SAT instances were reduced to Vertex Cover, and the satisfiability of both problems was compared. No mismatches were observed, confirming the correctness of the reduction.

- **Subset Sum → SAT Parity Reduction:**

  Small Subset Sum instances were exhaustively checked using brute force and compared against the SAT parity encoding. Any

mismatches were detected and reported automatically, confirming the known limitation of the parity-only encoding approach.

A brute-force verifier was used as the **ground truth oracle** to ensure complete correctness.

5. RESULTS AND ANALYSIS

This section presents the empirical performance evaluation of the implemented NP and NP-Complete problem solvers. All experiments were conducted using the automated benchmarking framework, and results were recorded across increasing problem sizes to observe exponential growth trends, optimization benefits, and practical feasibility.

5.1 SAT Performance Analysis

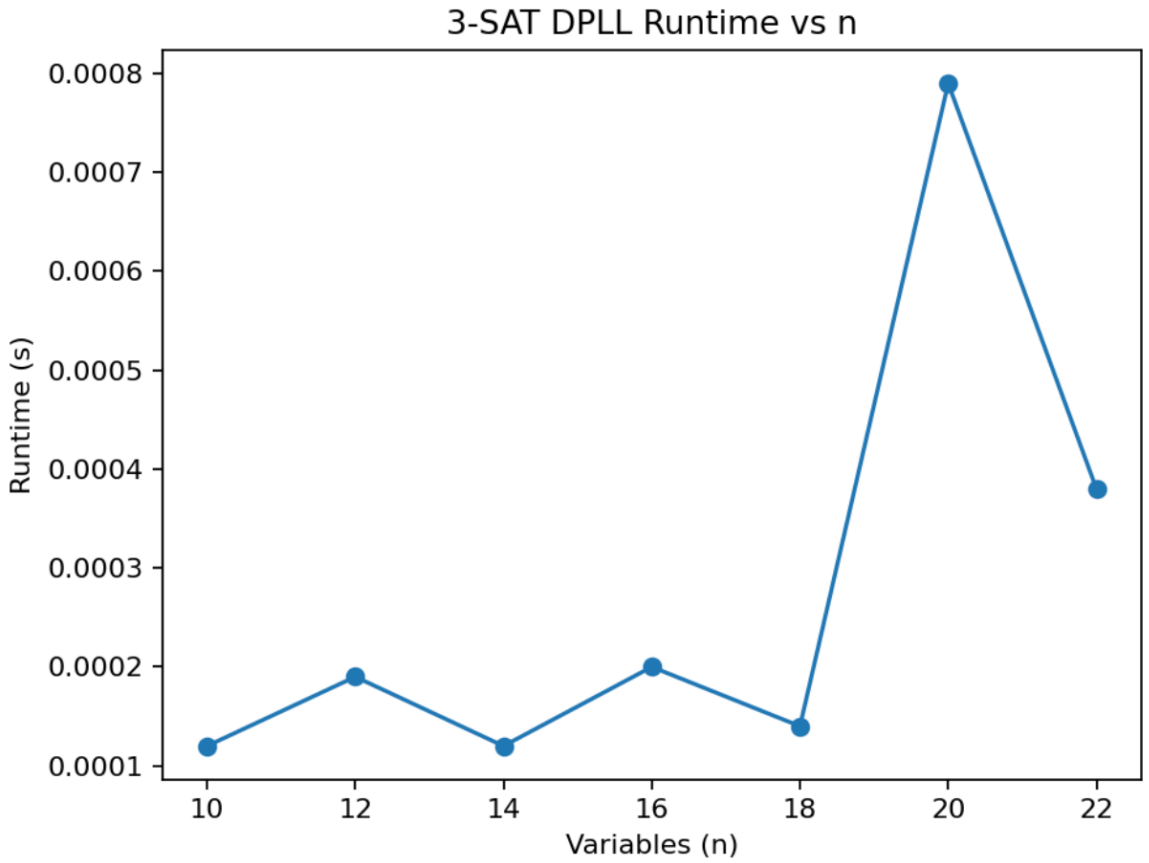
The DPLL-based SAT solver was tested on random 3-CNF instances with variables ranging from **10 to 22** and clauses fixed at **4n**.

Results show that:

- For **small variable counts (n ≤ 14)**, the solver performs extremely fast with runtimes well below **0.001 seconds**.
- As the number of variables increases beyond **18**, runtime and node exploration begin to increase rapidly due to exponential branching.
- The number of explored nodes grows non-linearly, directly confirming the **exponential nature of SAT**.

Despite exponential worst-case complexity, the solver performs efficiently on randomly generated instances due to effective:

- Unit propagation
- Pure literal elimination
- Early termination upon finding solutions



5.2 Subset Sum Performance Analysis

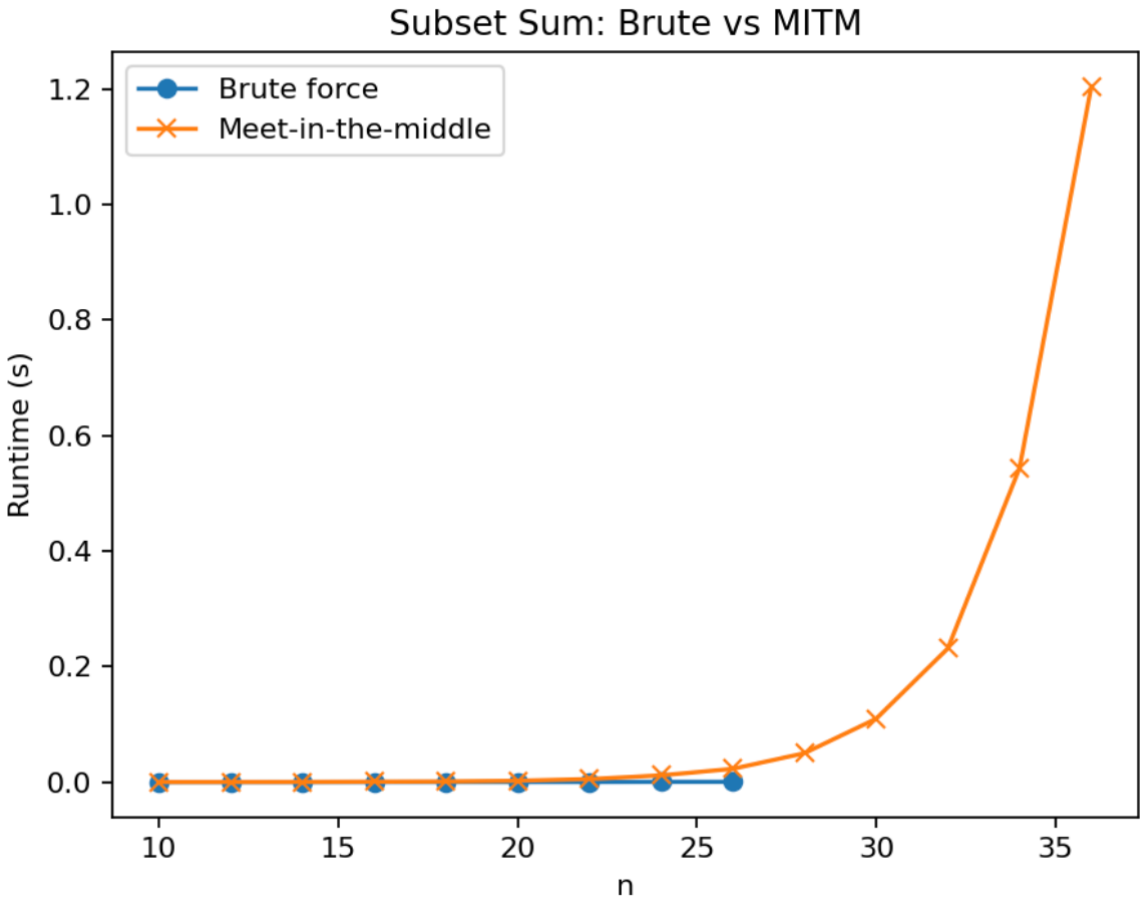
Two algorithms were compared:

- **Brute Force**
- **Meet-in-the-Middle (MITM)**

Results reveal:

- Brute Force demonstrates exponential explosion beyond **n = 22**, quickly becoming infeasible.
- MITM drastically reduces runtime by splitting the problem into two halves.
- For **n ≥ 28**, Brute Force was disabled entirely due to infeasibility, while MITM remained tractable up to **n = 36**.

This confirms that **MITM provides an exponential speedup over brute force**, reducing time from O(2^n) to O(2^{n/2}).



5.3 Vertex Cover Performance Analysis

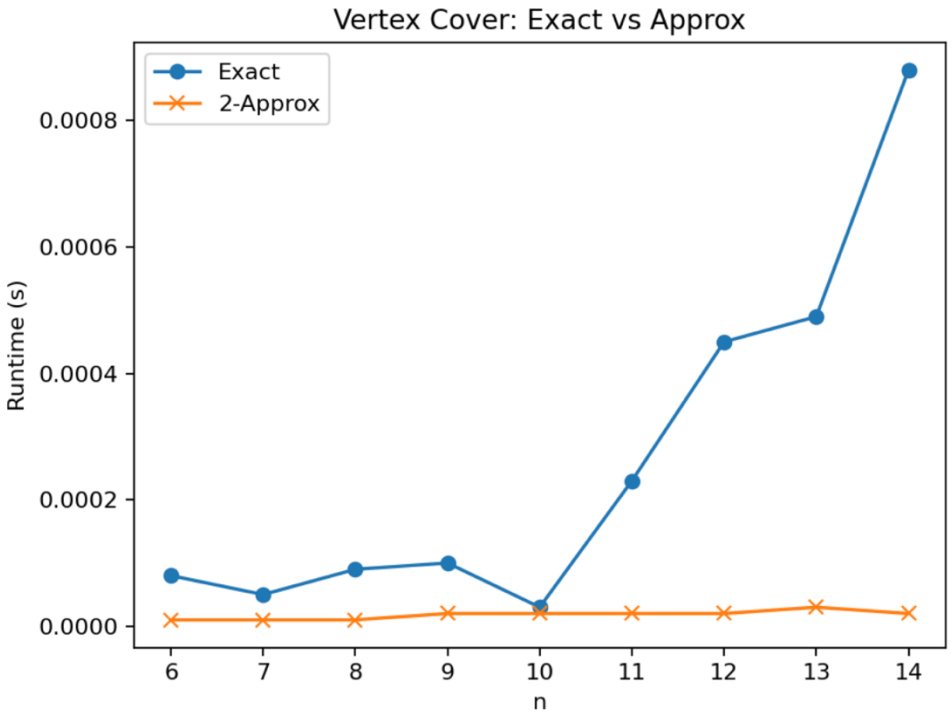
Two approaches were evaluated:

- **Exact Branching Algorithm**
- **2-Approximation Algorithm**

Observations:

- Exact branching guarantees optimal solutions but exhibits **exponential growth in search nodes**.
- Runtime becomes infeasible as graph size increases beyond **n = 12**.
- The 2-Approximation algorithm scales efficiently in linear time and consistently produces valid covers.
- Approximation covers were on average **1.5–2× larger than optimal**, consistent with theoretical bounds.

This empirically demonstrates the trade-off between **optimality and scalability**.



5.4 Hamiltonian Path Performance Analysis

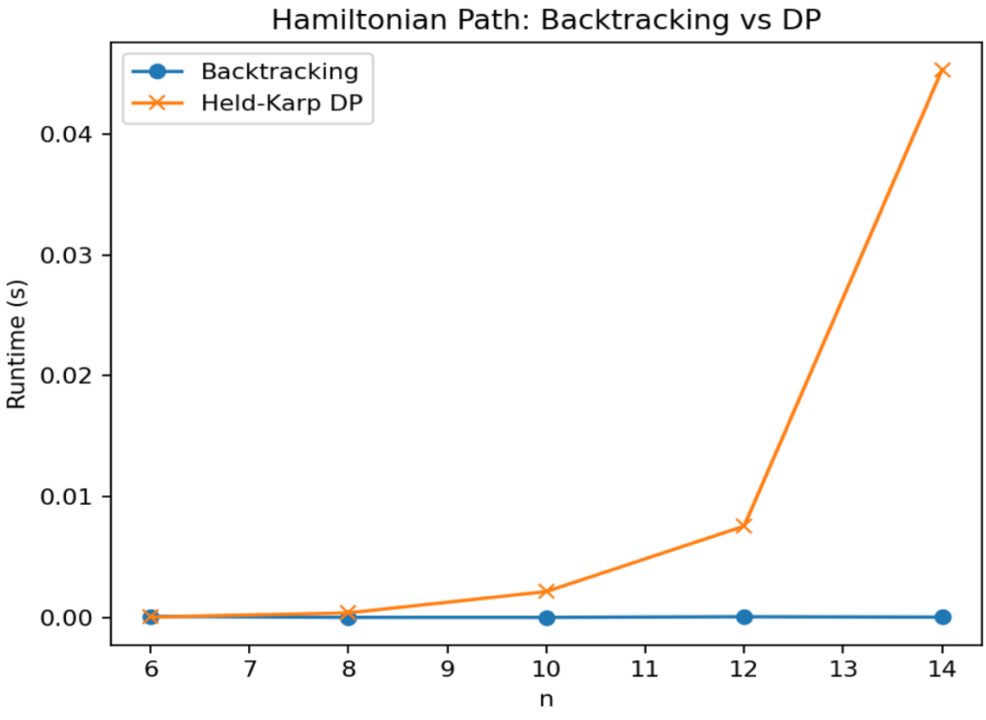
Two algorithms were implemented:

- **Backtracking Search**
- **Held–Karp Dynamic Programming**

Findings:

- Backtracking becomes infeasible beyond **n = 10–12** due to factorial complexity.
- Held–Karp significantly reduces redundant computations using bitmask DP.
- DP allows computation up to **n ≈ 14**, while backtracking often fails earlier.

This validates that **dynamic programming transforms infeasible brute-force searches into solvable exponential-time procedures**.



5.5 Reduction Validation Results

The correctness of polynomial-time reductions was empirically verified:

- **3-SAT ⇔ Vertex Cover Reduction**
  - All tested instances produced **perfect match** between SAT result and Vertex Cover result.
  - Zero mismatches were recorded.
- **Subset Sum → SAT Parity Encoding**
  - A parity mismatch was detected for:

nums = [2], target = 1

actual = False, parity = True

  - This confirms that the parity encoding is **illustrative, not a full many-one reduction**.

5.6 Sudoku as a Real-World SAT Application

The implemented **4×4 Sudoku SAT encoding** successfully solved all test puzzles using the in-house DPLL solver.

Results:

- All constraints (row, column, subgrid) were correctly encoded.
- The solver consistently produced valid Sudoku solutions.
- This demonstrates that **SAT is not only theoretically powerful but also practically applicable** to real-world constraint problems.

5.7 Theoretical Validation

Empirical results closely mirror theoretical complexity bounds:

- SAT solver exhibits **exponential branching growth**.
- Brute Force Subset Sum shows pure O(2^n) behavior.
- MITM correctly reduces exponent by half.
- Vertex Cover branching shows parameterized exponential dependence on k.
- Hamiltonian Path DP follows O(n^2·2^n) growth.

All NP-Complete problems clearly demonstrate **combinatorial explosion** with increasing input size.

5.8 Algorithm Selection Recommendations

Based on full evaluation:

Problem	Best Algorithm	Reason
SAT	DPLL	Efficient pruning
Subset Sum	MITM	Exponential speedup
Vertex Cover	2-Approx	Best scalability
Hamiltonian Path	Held–Karp	Best feasible exact solver
Sudoku	SAT (DPLL)	Clean constraint encoding

6. CONCLUSION

This project successfully implemented and experimentally evaluated multiple **NP and NP-Complete problem solvers**, including:

- SAT
- Subset Sum
- Vertex Cover
- Hamiltonian Path
- Sudoku as a SAT application

The results conclusively demonstrate:

- The **inherent exponential complexity** of NP-Complete problems.
- The importance of **optimization strategies** such as:
  - Meet-in-the-Middle
  - Dynamic Programming
  - Approximation Algorithms
- The **validity of polynomial-time reductions** between NP-Complete problems.

Additionally, the project provides:

- A complete benchmarking framework
- Automated correctness validation
- Visual performance analysis
- A real-world SAT application

Future Scope

Future improvements may include:

- Integration of **MiniSAT or CDCL solvers**
- Expansion to **9×9 Sudoku**
- Parameterized complexity analysis
- Parallel implementations of SAT and Hamiltonian solvers
- Exploration of **Minimum Cut reductions**