VNUHCM-UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

LAB: GEM HUNTER
COURSE: AI FOUNDATION

Lecturer:

Nguyễn Ngọc Đức

Nguyễn Trần Duy Minh

Nguyễn Thanh Tình

Student:

Hoàng Hồ Nhật Minh (22120208 – CQ2022/1)

Hồ Chí Minh, ngày 12 tháng 05 năm 2025

MỤC LỤC

I. Introduction:

Link Github: https://github.com/donalmun/Gem-hunter

Link Video: https://youtu.be/TeWln-84KWs

The Gem Hunter is a logic puzzle on a grid, where the player must identify the locations of the gems and avoid traps based on numeric clues. Each numbered cell indicates the number of traps in its 8 neighboring cells. The task is to model the problem as a set of CNF constraints and solve it using logic-based algorithms and SAT solvers.

Lab Objective:

The goal of this lab is to model the problem as a propositional logic system in CNF, solve it using the PySAT library, and compare the performance with brute-force and backtracking algorithms.

II. Logical Formulation & CNF:

1. Variable Mapping

Each unknow grid cell (r,c) (marked _) is assigned a unique positive integer variable.

- **True** value: The cell is a **trap**.
- **False** value: The cell is a **gem**.

This mapping is managed internally, allowing conversion between cell coordinates and variable integers.

2. Constraint Encoding for Numbered Cells:

When a cell in the grid contains a number, say N, this number imposes a constraint on its neighboring cells. Specifically, it means that among unknown cells adjacent to this numbered cell, exactly N of them must be traps. The system first identifies all valid neighboring positions for the numbered cell.

3. CNF Clause Generation:

The "exactly N traps among K unknown neighbors" rule is systematically converted into a set of CNF clauses. This is achieved by combining two distinct logical conditions, translated into clauses using combinations of the variables representing the K unknown neighbors:

- "At Least N Traps": Clauses ensuring a minimum of N traps. These are formed from combinations of K-N+1 positive neighbor variables.
- "At Most N Traps": Clauses ensuring a maximum of N traps. These are formed from combinations of N+1 negated neighbor variables.

4. Duplicate Clause Removal:

To optimize, the final set of CNF clauses is processed to remove any identical clauses, ensuring each unique logical constraint is represented only once. This often involves standardizing clause format (e.g., sorting literals) before adding to a set.

III. Automatic CNF Generation from Input:

1. Program Overview:

The program automates the entire process of converting a Gem Hunter puzzle, provided in a specific file format, into a solvable CNF formula. This involves parsing the input, establishing logical variables for unknown parts of the grid, and then systematically generating all necessary CNF clauses based on the numeric clues.

2. Input Processing Steps:

- **File Parsing:** The input file, representing the grid with numbers for clues and _ for unknowns (comma-space separated), is read and parsed into a 2D list structure. Numeric strings become integers.
- **Grid Initialization & Variable Mapping:** An internal grid representation is created. All unknown cells (_) are then systematically mapped to unique integer variables (as per Section 2.1).
- **Clause Generation from Clues:**
  - The system iterates through each cell of the initialized grid.
  - If a cell (i,j) contains an integer (a numeric clue, N), it first identifies all valid neighbouring cells.
  - For each valid neighbour that is an unknown cell (_), its corresponding pre-assigned integer variable is retrieved.
  - If there are any such unknown neighbour variables, the mechanism described in "Conversion to CNF Clauses" (Part 2) is invoked with this list of neighbour variables and the clue N to produce the CNF clauses for that specific clue.
  - These generated clauses are progressively added to a main CNF formula object.
- **Finalization:** After processing all cells and their clues, the accumulated set of clauses undergoes duplicate removal (as described in Part 2) to produce the final, optimized CNF formula representing the entire puzzle**.**

3. Example CNF:

Consider a 3x3 input grid with a clue 1 at cell (0,1). Let its 5 unknown neighbors have been mapped to integer variables V1, V2, V3, V4, V5 during the variable mapping phase.The clue (0,1)=1 dictates "exactly one of V1, V2, V3, V4, V5 is True (a trap)".This is translated into CNF clauses by combining:

- "At Least 1 Trap": Clauses ensuring at least one of V1...V5 is true. For example, a clause (V1 ∨ V2 ∨ V3 ∨ V4 ∨ V5) (using DIMACS-like notation where positive integers are variables and 0 terminates the clause). This is formed from combinations of 5-1+1 = 5 variables.
- "At Most 1 Trap": Clauses ensuring no more than one of V1...V5 is true. This involves creating clauses from all combinations of 1+1 = 2 variables, where both are negated. For example:

- ¬V1 ∨ ¬V2 (meaning V1 and V2 cannot both be traps)
- ¬V1 ∨ ¬V3
- ... and so on for all pairs from V1...V5.

The complete CNF for an entire grid would aggregate such sets of clauses derived from every numeric clue present.

IV. Solution Algorithms:

1. Solving with PySAT Library:

a) Operating Principle:

This method leverages the capabilities of the PySAT library, a high-performance SAT (Satisfiability) solver. Instead of implementing complex search logic, the program delegates the task of solving the CNF formula to this external library.

b) Implementation Idea:

CNF Handover: The complete puzzle is passed to a PySAT solver instance.
Solving: The library's *solve()* method is called, applying advanced SAT techniques.
Result:

- If satisfiable, a "model" (list of positive/negative integers representing True/False variable assignments) is returned.
- If unsatisfiable, no solution is reported.

2. Brute-force Algorithm:

a) Operating Principle:

The brute-force algorithm systematically explores every possible combination of assignments (Trap or Gem) for each unknown cell in the grid to find a solution.

b) Implementation Idea:

Combination Generation: Iterates through all 2^n truth assignments (where n is #variables), typically using the binary representation of integers from 0 to $2^n - 1$.
Attempt Limit: Capped at a maximum of attempts (e.g., 2M) to manage runtime.
Satisfiability Checking: For each generated assignment combination:

- A utility function (from the CNF handling module) is employed to check if the current assignment combination satisfies all clauses in the stored CNF formula (i.e., makes them all True).
- If an assignment combination that satisfies all clauses is found, it is converted into the standard "model" format and returned as the solution.
- If no satisfying combination is found after all possibilities have been tried (or the limit is reached), the algorithm concludes that there is no solution.

3. Backtracking Algorithm (DPLL-style)

a) Operating Principle:

Employs a recursive depth-first search, similar to the DPLL algorithm, to construct a solution incrementally, backtracking when a path becomes invalid.

b) **Implementation Idea:**

Recursive Structure: A core recursive function takes the current (potentially simplified) CNF and a partial variable assignment.

Attempt Limit: A recursion depth limit (e.g., 2M) controls execution time.

Optimization – Unit Propagation:

- Identifies "unit clauses" - clauses with only one unassigned literal left
- Makes forced assignments: when a unit clause is found, its variable must be assigned to make that literal true
- Simplifies the CNF by removing satisfied clauses and false literals from remaining clauses
- Detects contradictions early: if an empty clause appears, backtracks immediately
- Significantly improves efficiency by reducing unnecessary branching

Recursion Base Case:

- Success: Empty CNF (all clauses satisfied)
- Failure: Empty clause encountered (contradiction)
- Limit: Maximum recursion depth reached

Variable Selection & Branching:

- Selects an unassigned variable.
- Recursively tries assigning True, simplifying CNF. If it fails, backtracks.
- Recursively tries assigning False, simplifying CNF.
- If both branches fail, this path fails.

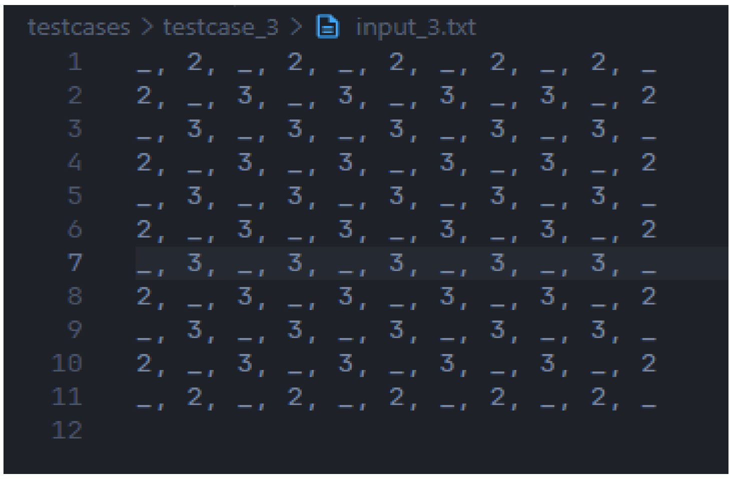
Result Conversion: Converts successful assignments to the standard model format (positive integers for traps, negative for gems).

V. Experiments & Results Analysis:

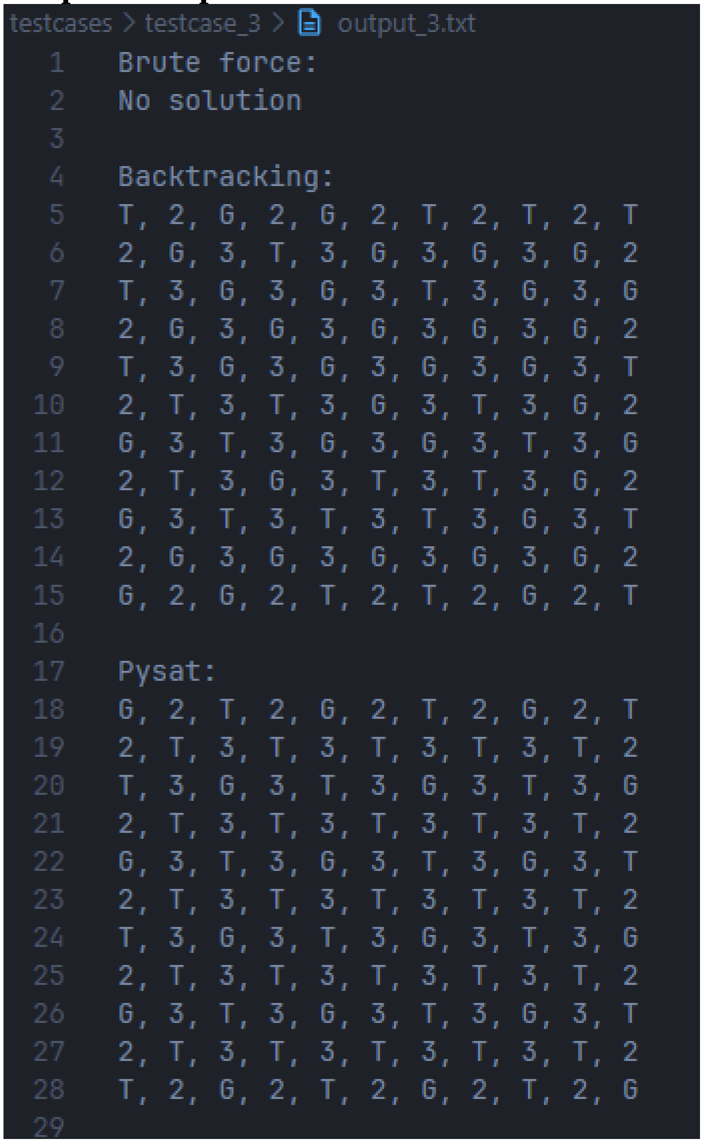
1. Test Case Setup:

A suite of 6 test cases, located in the testcases/ directory (specifically testcase_0 through testcase_5), was utilized. For each test case, input files (input_x.txt) define the puzzle grid, and corresponding output files (output_x.txt) are generated by each algorithm to store the solution and verify correctness. The execution times for each algorithm on each test case were systematically recorded.

Input example:


```
testcases > testcase_3 > input_3.txt
_, 2, _, 2, _, 2, _, 2, _, 2, _
2, _, 3, _, 3, _, 3, _, 3, _, 2
_, 3, _, 3, _, 3, _, 3, _, 3, _
2, _, 3, _, 3, _, 3, _, 3, _, 2
_, 3, _, 3, _, 3, _, 3, _, 3, _
2, _, 3, _, 3, _, 3, _, 3, _, 2
_, 3, _, 3, _, 3, _, 3, _, 3, _
2, _, 3, _, 3, _, 3, _, 3, _, 2
_, 3, _, 3, _, 3, _, 3, _, 3, _
2, _, 3, _, 3, _, 3, _, 3, _, 2
_, 2, _, 2, _, 2, _, 2, _, 2, _
```

Output example:

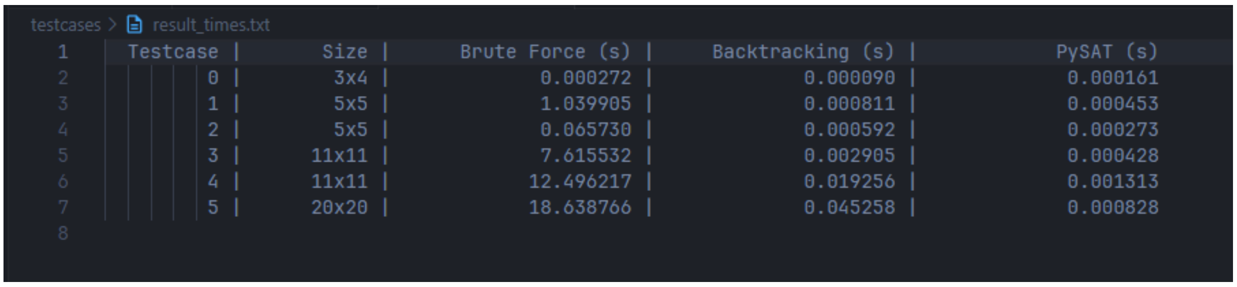

```
testcases > testcase_3 > output_3.txt
Brute force:
No solution

Backtracking:
T, 2, G, 2, G, 2, T, 2, T, 2, T
2, G, 3, T, 3, G, 3, G, 3, G, 2
T, 3, G, 3, G, 3, T, 3, G, 3, G
2, G, 3, G, 3, G, 3, G, 3, G, 2
T, 3, G, 3, G, 3, G, 3, G, 3, T
2, T, 3, T, 3, G, 3, T, 3, G, 2
G, 3, T, 3, G, 3, G, 3, T, 3, G
2, T, 3, G, 3, T, 3, T, 3, G, 2
G, 3, T, 3, T, 3, T, 3, G, 3, T
2, G, 3, G, 3, G, 3, G, 3, G, 2
G, 2, G, 2, T, 2, T, 2, G, 2, T

Pysat:
G, 2, T, 2, G, 2, T, 2, G, 2, T
2, T, 3, T, 3, T, 3, T, 3, T, 2
T, 3, G, 3, T, 3, G, 3, T, 3, G
2, T, 3, T, 3, T, 3, T, 3, T, 2
G, 3, T, 3, G, 3, T, 3, G, 3, T
2, T, 3, T, 3, T, 3, T, 3, T, 2
T, 3, G, 3, T, 3, G, 3, T, 3, G
2, T, 3, T, 3, T, 3, T, 3, T, 2
G, 3, T, 3, G, 3, T, 3, G, 3, T
2, T, 3, T, 3, T, 3, T, 3, T, 2
T, 2, G, 2, T, 2, G, 2, T, 2, G
```

2. Recorded Performance Data:

The execution times (in seconds) for each algorithm across the test cases, as recorded in testcases/static_times.txt, are summarized below:



```
Testcase |        Size |      Brute Force (s) |      Backtracking (s) |             PySAT (s)
       0 |         3x4 |             0.000272 |              0.000090 |              0.000161
       1 |         5x5 |             1.039905 |              0.000811 |              0.000453
       2 |         5x5 |             0.065730 |              0.000592 |              0.000273
       3 |       11x11 |             7.615532 |              0.002905 |              0.000428
       4 |       11x11 |            12.496217 |              0.019256 |              0.001313
       5 |       20x20 |            18.638766 |              0.045258 |              0.000828
```

3. Analysis of Results:

a) Correctness:

Testcase	**Brute Force(s)**	**Backtracking(s)**	**PySAT(s)**
0	**0.000272**	**0.000090**	**0.000161**
1	**1.039905**	**0.000811**	**0.000453**
2	**0.065730**	**0.000592**	**0.000273**
3	**Exceeded max attempts**	**0.002905**	**0.000428**
4	**Exceeded max attempts**	**0.019256**	**0.001313**
5	**Exceeded max attempts**	**0.045258**	**0.000828**

- **Small Grid (3x4):**
  - All three algorithms found correct solutions, with each numeric clue's constraint properly satisfied.
- **Medium Grids (5x5):**
  - All algorithms solved both 5x5 test cases correctly.
  - Brute Force time varied significantly (1.04s vs 0.07s) between the two cases, showing puzzle configuration impacts difficulty.
- **Large Grids (11x11 and 20x20):**
  - Brute Force reached maximum attempt limit and failed to solve these instances.
  - Backtracking and PySAT both found valid solutions for all large grids.
  - Though different solutions were found by each algorithm, all satisfied the game's constraints completely.

b) Analysis:

- **PySAT** is consistently the fastest algorithm across all test cases. Its specialized algorithms and optimizations deliver exceptional performance, maintaining sub-millisecond solving times even for 20x20 grids. This confirms why dedicated SAT solvers excel at complex constraint problems.
- **Backtracking with Unit Propagation** significantly outperforms Brute-force by efficiently pruning the search space. By identifying forced assignments and detecting contradictions early, it successfully handles large grids that Brute-force cannot solve. While slower than PySAT, it offers a good balance between performance and implementation simplicity.
- **Brute-force** shows clear limitations due to its exponential complexity.

Performance degrades rapidly with grid size, making it impractical for anything beyond small puzzles. It serves primarily as a baseline reference or educational example rather than a practical solution for real-world instances.

VI. Process:

No.	Criteria	Done
1	**Solution description:** Describe the correct logical principles for generating CNFs.	100%
2	Generate CNFs automatically	100%
3	Use pysat library to solve CNFs correctly	100%
4	Program brute-force algorithm to compare with using library(speed)	100%
5	Program backtracking algorithm to compare with using library (speed)	100%
6	**Documents and other resources that you need to write and analysis in your report:** • Thoroughness in analysis and experimentation • Give at least 3 test cases with different sizes (5x5, 11x11, 20x20) to check your solution • Comparing results and performance	100%

VII. References:

- PySat Documentation: https://pysathq.github.io/
- Davis-Putnam-Logemann-Loveland (DPLL) Algorithm: DPLL algorithm - Wikipedia
- SAT Competition (benchmarks and solvers): www.satcompetition.org