HO CHI MINH CITY UNIVERSITY OF SCIENCE

Artificial Intelligence

PROJECT: GEM HUNTER

Author: Pham Le Thai Bao

Student ID: 23127159

Class: 23CLC07

April 6, 2025

REPORT PROJECT: GEM HUNTER

Pham Le Thai Bao

April 6, 2025

Contents

§1 Abstract

Abstract : This report details the implementation and performance evaluation of three distinct algorithms for solving the Gem Hunter logic puzzle: a dedicated SAT solver library (PySAT), a custom Backtracking algorithm, and a Brute Force search. The puzzle's rules are systematically encoded into Conjunctive Normal Form (CNF), allowing the problem to be treated as a Boolean Satisfiability (SAT) problem. Each algorithm attempts to find a satisfying assignment for the CNF formula, which corresponds to a valid placement of traps ('T') and safe gems ('G') on the puzzle grid. Experiments were conducted on three test cases of increasing size (5x5, 11x11, 20x20). Performance was primarily measured by execution time. The results demonstrate the high efficiency of the PySAT library, which significantly outperformed the Backtracking algorithm, especially on larger grids. The Brute Force algorithm proved computationally infeasible for all but the smallest test case due to its exponential complexity. This study highlights the effectiveness of using CNF encoding and dedicated SAT solvers for combinatorial problems like Gem Hunter.

§2 Conjunctive Normal Form (CNF) for Gem Hunter

Boolean Satisfiability (SAT)

The core problem being addressed is SAT. Given a Boolean formula constructed from variables, logical AND ($\wedge$), logical OR ($\vee$), and negation ($\neg$), the SAT problem asks if there is an assignment of **True** or **False** values to each variable that makes the entire formula evaluate to **True**.

Literal (Detailed Definition for Gem Hunter)

In Boolean logic, a **literal** is the most basic element representing a proposition or its inverse. It consists of a Boolean variable or the negation of that variable.

In this Project: Each potentially unknown cell (an empty cell `_` in the input grid) at coordinates (row, col) is represented by a unique Boolean variable. The function `helper.to_1D` maps these grid coordinates (row, col) to a unique positive integer v. This integer v serves as the identifier for the Boolean variable associated with that cell.

- **Positive Literal (v):** The positive integer v itself is used as a positive literal. If this literal v is assigned the value `True` in a potential solution, it signifies the proposition: *"The cell at coordinates (row, col) contains a trap ('T')"*.
- **Negative Literal ($\neg v$ or $-v$):** The negation of the variable, represented in the code and standard DIMACS CNF format as the negative integer $-v$, is a negative literal. If this literal $-v$ is assigned the value `True` (which implies the variable v itself is `False`), it signifies the proposition: *"The cell at coordinates (row, col) does not contain a trap ('T')"*, meaning it must contain a safe gem ('G').

Therefore, every empty cell gives rise to two possible literals: v (trap) and $-v$ (no trap/gem). Finding a solution involves assigning `True` or `False` to each underlying variable v.

Conjunctive Normal Form (CNF) (Detailed Definition for Gem Hunter)

Structure: A Boolean formula is in Conjunctive Normal Form (CNF) if it is structured as a conjunction (logical AND, $\wedge$) of one or more clauses. Each clause, in turn, is a disjunction (logical OR, $\vee$) of one or more literals. This creates an *"AND of ORs"* structure:

$$(Literal_{11} \vee Literal_{12} \vee \ldots) \wedge (Literal_{21} \vee Literal_{22} \vee \ldots) \wedge \ldots$$

Significance: CNF is crucial because it's the standard input format required by the vast majority of efficient SAT solvers, including the ones used in the `PySAT` library. The challenge is to translate the specific rules and constraints of a problem (like Gem Hunter) into this standardized CNF structure.

Encoding Gem Hunter Constraints: The core logic of Gem Hunter lies in the numbered cells. If a cell contains a number k, it means exactly k of its adjacent cells must be traps. This "exactly k" constraint must be translated into CNF.

Let $N = \{v_1, v_2, \ldots, v_n\}$ be the set of variables (integer IDs) corresponding to the n empty neighboring cells of the numbered cell k.

The constraint **"exactly k traps among N"** is broken down into two parts, both of which must be true (hence they will be ANDed together in the final CNF):

1. **At Most k Traps:** This means it's impossible to have $k+1$ or more traps among the neighbors. This is enforced by stating that for every possible subset of $k+1$ neighboring variables, **at least one of them must not be a trap** (must be False). Choosing a subset $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{k+1}}\} \subseteq N$ leads to the clause:

$$(\neg v_{i_1} \vee \neg v_{i_2} \vee \ldots \vee \neg v_{i_{k+1}})$$

   In the code (`generateExactTrapsClauses`), this corresponds to generating clauses like `[-x for x in subset]` for all subsets of size $k+1$.

2. **At Least k Traps:** This means it's impossible to have fewer than k traps, or equivalently, it's impossible to have $n-k+1$ or more non-traps (gems) among the neighbors. This is enforced by stating that for every possible subset of $n-k+1$ neighboring variables, **at least one of them must be a trap** (must be True). Choosing a subset $\{v_{j_1}, v_{j_2}, \ldots, v_{j_{n-k+1}}\} \subseteq N$ leads to the clause:

$$(v_{j_1} \vee v_{j_2} \vee \ldots \vee v_{j_{n-k+1}})$$

   In the code, this corresponds directly to the subsets generated for the "at least" case (clauses of positive literals).

Final Knowledge Base (KB): The `KB` variable in the code (`solvers.py`, generated by `CNF.py::getCNFClause`) represents the complete CNF formula for the entire puzzle. It is a list (representing the top-level AND conjunction) where each element is a list of integers (representing a clause, with the integers being the literals joined by OR). This `KB` is the logical AND of all the "at most k" and "at least k" clauses generated from every numbered cell in the grid.

Satisfying the CNF: A truth assignment to the variables (empty cells) satisfies the CNF formula (`KB`) if and only if it makes every single clause in the `KB` evaluate to `True`. This directly corresponds to fulfilling all the numerical constraints of the Gem Hunter puzzle simultaneously.

§3 Algorithms

§3.1 Brute Force

The Brute Force algorithm generates all possible 2^N assignments (where N is the number of Boolean variables) and checks each assignment using the `isValid` function to determine if it satisfies the knowledge base `KB`.

- This method is implemented in the file `bruteforce.py`.
- If the number of variables N is too large (exceeding the`BRUTEFORCE_LIMIT` threshold), the Brute Force method will be skipped by `table.py`.
- This method is computationally infeasible for non-trivial game grids due to the exponential increase in time complexity.

§3.2 Backtracking

The Backtracking algorithm also explores the space of all assignments for N Boolean variables (representing empty cells). However, unlike Brute Force, it does not exhaustively check all assignments. Instead, it prunes invalid branches early using the isConflict() function, avoiding unnecessary checks of assignments that are guaranteed to fail.

- This method is implemented in the file backtracking.py.
- While the theoretical complexity remains $O(2^N)$, in practice, `backtracking` is significantly more efficient than Brute Force thanks to its ability to prune unfeasible branches early.

§3.3 Pysat

The PySAT algorithm converts the knowledge base (KB) into Conjunctive Normal Form (CNF) and uses an industrial-strength SAT solver to find a solution. The solver applies advanced techniques like unit propagation, clause learning, and conflict-driven search to efficiently explore the solution space. If a valid assignment of variables exists that satisfies all clauses, the solver returns it as a model.

- The PySAT algorithm uses an industrial-grade SAT solver to efficiently find a satisfying assignment for the given knowledge base KB (expressed in CNF form).
- This method is implemented in the file `pysat.py`.
- The knowledge base `KB` is first converted into the CNF format supported by the `PySAT` library.
- The solver explores the solution space using advanced SAT solving techniques (such as clause learning, unit propagation, conflict-driven search, etc.), which are much faster and more optimized than Brute Force or Backtracking for complex problems.
- If a solution exists, the solver returns a model that assigns truth values to variables, satisfying all clauses in KB.
- This method is highly efficient and is suitable for large-scale or complex game grids, where Brute Force or Backtracking would be computationally infeasible.

§4 Experiments

§4.1 Testcases

The speeds of the three algorithms were evaluated using the following three test cases:

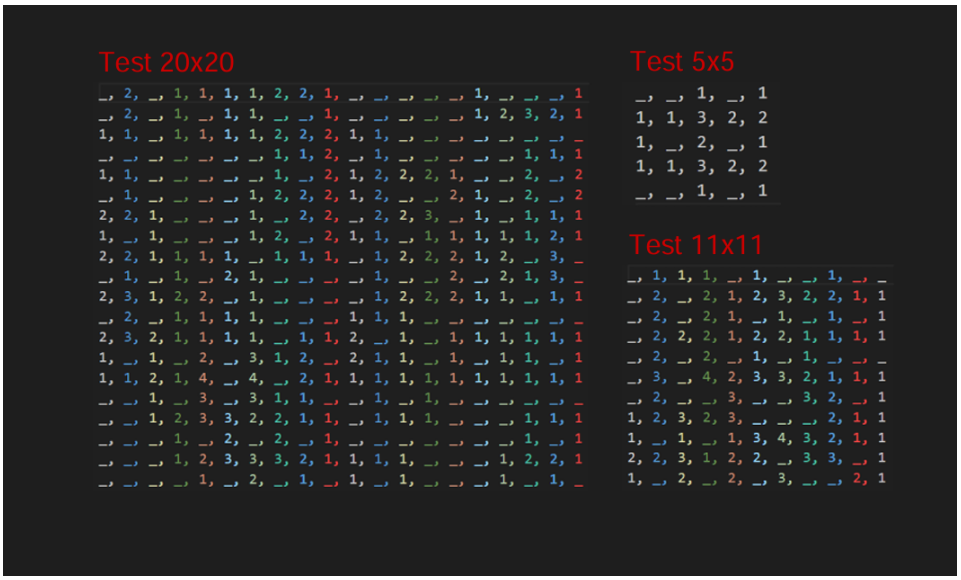

Figure 1: Testcases for Experiment

§4.2 Result

§4.2.1 Brute Force

- The algorithm of the **Brute Force** solver generates outputs:

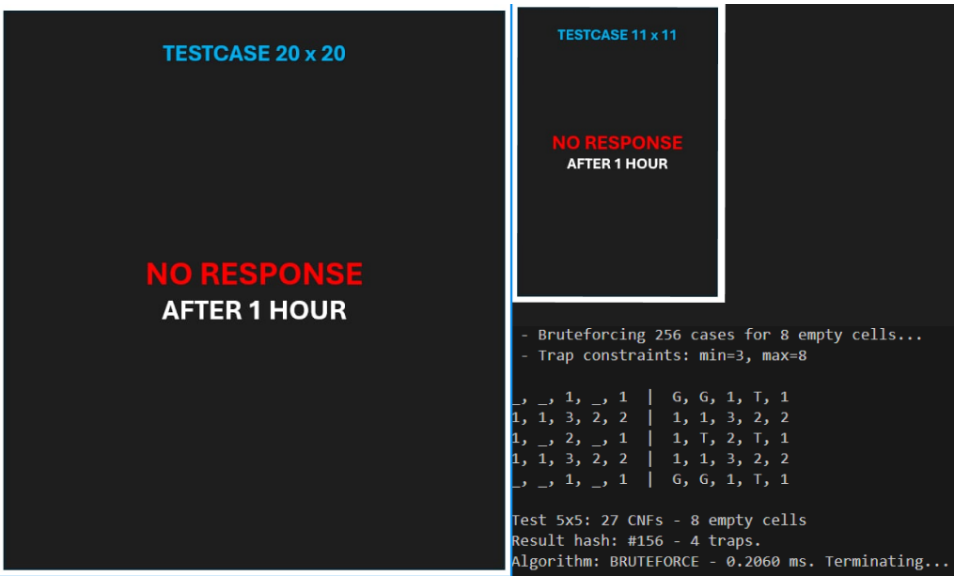

Figure 2: Outputs generated from Brute Force algorithm

- The runtime of algorithm for three testcases was recorded:
  - **Testcase 5x5**: 0.2060 ms.
  - **Testcase 11x11**: $>$ 1 hour.
  - **Testcase 20x20**: $>$ 1 hour.
- The experimental results show that the Brute Force algorithm performs well on very small instances, such as the 5x5 grid, solving it in just 0.2060 ms. However, as the problem size increases, the performance degrades drastically. For both the 11x11 and 20x20 grids, the algorithm takes over 1 hour to compute, clearly demonstrating the exponential growth in computation time.

  $\rightarrow$ This confirms that Brute Force is only practical for very small grids and becomes infeasible for larger, more complex instances.

§4.2.2 Backtracking

- The **Backtracking** algorithm generates outputs:

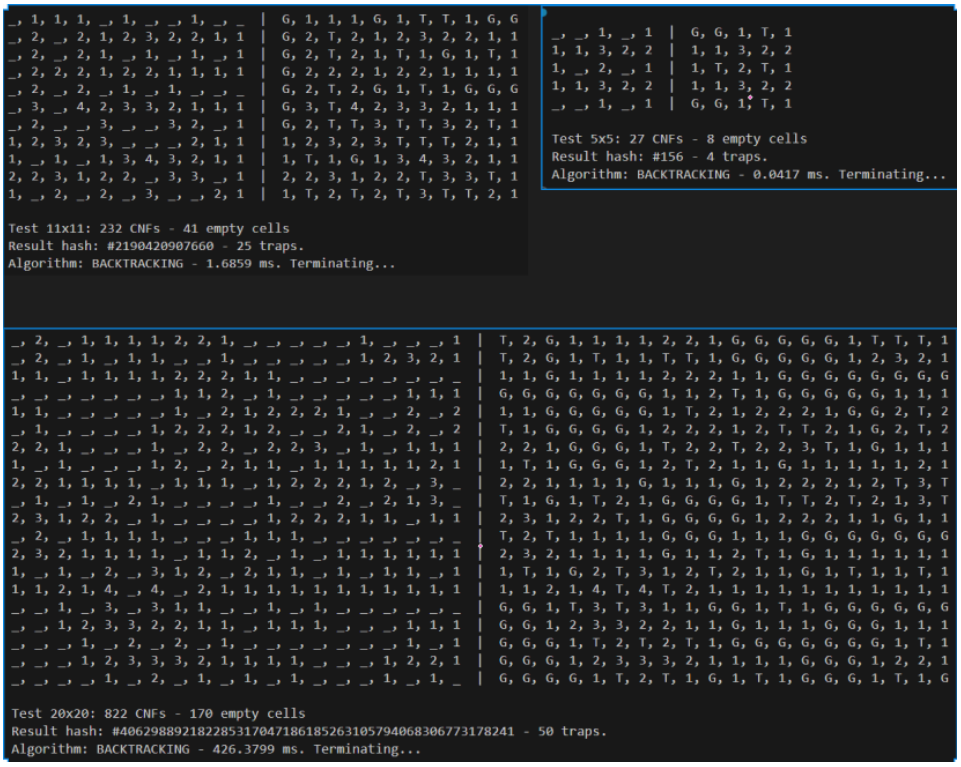

Figure 3: Outputs generated from Backtracking algorithm

- The runtime of algorithm for three testcases was recorded:
  - **Testcase 5x5**: 0.0417 ms.
  - **Testcase 11x11**: 1.6859 ms .
  - **Testcase 20x20**: 426.3799 ms.
- The experimental results demonstrate that the Backtracking algorithm scales much better than Brute Force. It solves the 5x5 grid in just 0.0417 ms and handles the larger 11x11 grid efficiently at 1.6859 ms. Even for the significantly larger 20x20

grid, the algorithm completes in 426.3799 ms, which, while higher, is still vastly faster compared to Brute Force.

→ These results highlight that Backtracking is a practical approach for moderate-sized problems, offering a good balance between completeness and performance.

§4.2.3 Pysat

- The algorithm of the **PySAT** solver generates outputs:
- The **Backtracking** algorithm generates outputs:

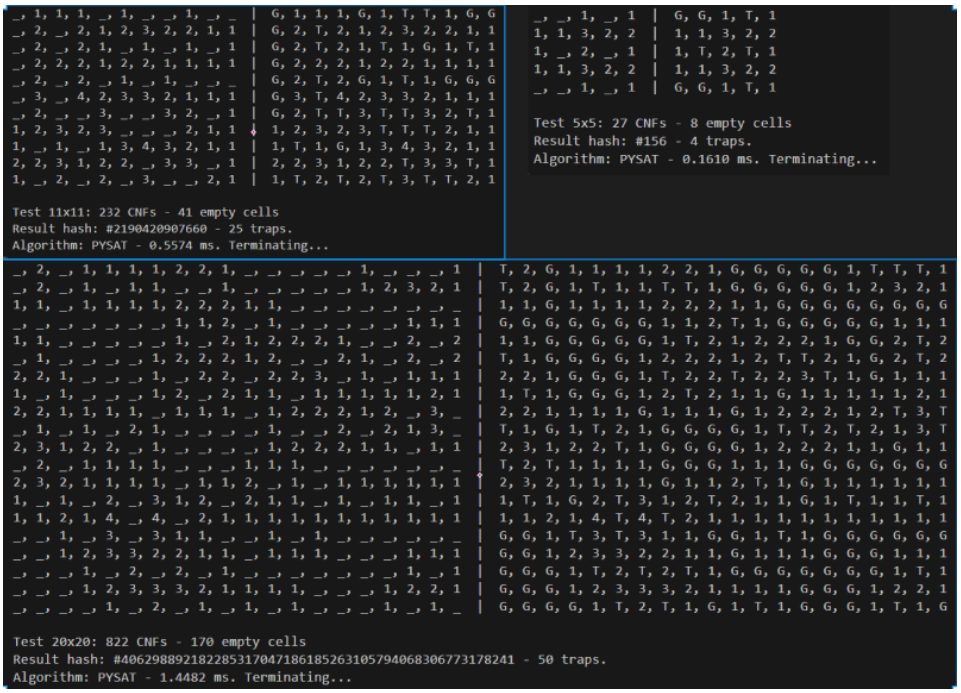

Figure 4: Outputs generated from **PySAT** algorithm

- The runtime of algorithm for three testcases was recorded:
  - **Testcase 5x5**: 0.1610 ms.
  - **Testcase 11x11**: 0.5574 ms .
  - **Testcase 20x20**: 1.4482 ms.
- The experimental results show that the PySAT-based algorithm performs exceptionally well across all test cases. It solves the 5x5 grid in just 0.1610 ms and the 11x11 grid in 0.5574 ms. Even for the larger 20x20 grid, the runtime remains very low at 1.4482 ms.

  → These results demonstrate that PySAT is highly efficient and scalable, making it an excellent choice for solving SAT problems, even as the grid size increases.

§4.2.4 Final experimental result

Statistic table

Test case	CNFs	Empty cells	Traps	Algorithm	Time	Model hash (binary)
5x5	27	8	4	pysat	0.1610 ms	156
5x5	27	8	4	backtracking	0.0417 ms	156
5x5	27	8	4	bruteforce	0.2060 ms	156
-	-	-	-	-	-	-
11x11	232	41	25	pysat	0.5574 ms	2190420907660
11x11	232	41	25	backtracking	1.6859 ms	2190420907660
11x11	232	41	25	bruteforce	N/A	2190420907660
-	-	-	-	-	-	-
20x20	822	170	50	pysat	1.4482 ms	406298892182285317047186185263105794068306773178241
20x20	822	170	50	backtracking	426.3799 ms	406298892182285317047186185263105794068306773178241
20x20	822	170	50	bruteforce	N/A	406298892182285317047186185263105794068306773178241

Figure 5: Final experimental result.

- Since the runtime of the Brute Force algorithm on the 11x11 and 20×20 testcases exceeds one hour, it is regarded as N/A for practical purposes.

§5 Report Completion Status Table

No.	Criteria	Processing
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, 10x10, 20x20) to check your solution Comparing results and performance	100%

§6 Demonstration Video

YouTube Link: https://youtu.be/ADkt1lX6FEY

§7 References

References

[1] https://pysathq.github.io/docs/html/index.html

[2] https://github.com/git03-Nguyen/Gem-Hunter-Solvers