VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION AND TECHNOLOGY

23CLC08
Lab: GEM HUNTER

INSTRUCTOR

Nguyen Ngoc Thao

Nguyen Thanh Tinh

Nguyen Tran Duy Minh

—o0o—

STUDENT

23127318 - Le Ho Dan Anh

HO CHI MINH CITY, APRIL 2025

Mục lục

1 Lab overall **2**
1.1 Level of completion . . . 2
1.2 How to run the program . . . 2
1.3 Demo video . . . 2

2 CNF Formulation **3**
2.1 Problem approach . . . 3
2.2 CNF Constraints . . . 3
2.2.1 Case 1: No traps around ($n = 0$) . . . 3
2.2.2 Case 2: All neighbors are traps ($n = |S|$) . . . 3
2.2.3 Case 3: Some but not all neighbors are traps ($0 < n < |S|$) . . . 4
2.3 Final CNF representation . . . 4

3 Solver analysis **5**
3.1 Brute force solver . . . 5
3.2 Backtracking solver . . . 5
3.3 PySAT solver (using Glucose3) . . . 5
3.4 Experimentation . . . 6
3.4.1 Result . . . 6
3.4.2 Performance analysis . . . 10
3.4.3 Summary . . . 10

1 Lab overall

1.1 Level of completion

No.	Criteria	Completion
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%

1.2 How to run the program

To run the program, follow these steps:

1. Install the required dependencies by running:

   ```
   pip install -r requirements.txt
   ```

2. Execute the program with the desired algorithm and test case by running:

   ```
   python main.py <algorithm> <test_case>
   ```

   - **Available algorithms:** bruteforce, backtracking, pysat, all
   - **Available test cases:** 5x5, 11x11, 20x20, 9x9

1.3 Demo video

2 CNF Formulation

2.1 Problem approach

- The grid is represented as a 2D array. *i,j* represents the position of a cell in row i and column j of the grid.
- Each numbered tile $N_{i,j}$ represents the exact count of traps (T) in its surrounding 8 neighboring cells.
- Each empty position $X_{i,j}$ in the grid is represented by a unique SAT variable, as follows:
  - T (Trap): Corresponds to the value True (1) for the SAT variable.
  - F (Gem): Corresponds to the value False (0) for the SAT variable

The goal is to express these conditions as CNF clauses of SAT variables, which are conjunctions (AND operations) of disjunctions (OR operations).

2.2 CNF Constraints

For a numbered cell $N_{i,j}$, let S be the set of its valid neighboring cells:

$$S = \{X_{k,l} \mid (k,l) \text{ is a neighbor of } (i,j)\}$$

Let $|S|$ be the number of neighbors of $N_{i,j}$, and let n be the value of $N_{i,j}$.

2.2.1 Case 1: No traps around ($n = 0$)

If a numbered cell contains 0, then all its neighboring cells must be gems:

$$\bigwedge_{X \in S} \neg X$$

In CNF form:

$$(\neg X_1) \wedge (\neg X_2) \wedge \cdots \wedge (\neg X_m)$$

2.2.2 Case 2: All neighbors are traps ($n = |S|$)

If n equals the total number of neighbors, then all neighbors must be traps:

$$\bigwedge_{X \in S} X$$

In CNF form:

$$(X_1) \wedge (X_2) \wedge \cdots \wedge (X_m)$$

2.2.3 Case 3: Some but not all neighbors are traps ($0 < n < |S|$)

For this case, we impose two constraints:

(a) At Most n Traps ($\leq n$): At most n neighbors are traps, meaning any subset of size $n+1$ must contain at least one non-trap:

$$\bigwedge_{T \subseteq S, |T|=n+1} \left(\bigvee_{X \in T} \neg X \right)$$

In CNF form:

$$(\neg X_{i_1} \vee \neg X_{i_2}) \wedge (\neg X_{i_1} \vee \neg X_{i_3}) \wedge \ldots$$

Example for $n = 2$, $S = \{X_1, X_2, X_3, X_4\}$:

$$(\neg X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_4) \wedge \ldots$$

(b) At Least n Traps ($\geq n$): At least n neighbors are traps, meaning any subset of size $|S| - n + 1$ must contain at least one trap:

$$\bigwedge_{T \subseteq S, |T|=|S|-n+1} \left(\bigvee_{X \in T} X \right)$$

In CNF form:

$$(X_{i_1} \vee X_{i_2}) \wedge (X_{i_1} \vee X_{i_3}) \wedge \ldots$$

Example for $n = 2$, $S = \{X_1, X_2, X_3, X_4\}$:

$$(X_1 \vee X_2) \wedge (X_1 \vee X_3) \wedge (X_2 \vee X_3) \wedge \ldots$$

2.3 Final CNF representation

The complete CNF constraints for a numbered tile $N_{i,j}$ are:

$$\begin{aligned}
&\text{If } n = 0: && \bigwedge_{X \in S} \neg X \\
&\text{If } n = |S|: && \bigwedge_{X \in S} X \\
&\text{If } 0 < n < |S|: && \begin{cases} \bigwedge_{T \subseteq S, |T|=n+1} (\bigvee_{X \in T} \neg X) & (\leq n) \\ \bigwedge_{T \subseteq S, |T|=|S|-n+1} (\bigvee_{X \in T} X) & (\geq n) \end{cases}
\end{aligned}$$

When implementating:

- Use the `set()` data structure to store CNF clauses to avoid *duplicate clauses.*
- The empty cells $X_{i,j}$ are mapped to SAT variables in a specific order, from top to bottom, left to right, as SAT variables $x_1, x_2, x_3, \ldots$
- Each SAT variable $x_1, x_2, x_3, \ldots$ is represented by an integer.
  - A positive integer (e.g., $1, 2, 3, \ldots$) represents the SAT variable in the true state, while a negative integer (e.g., $-1, -2, -3, \ldots$) represents the SAT variable in the false state (which is the negation of the variable).

3 Solver analysis

3.1 Brute force solver

- Brute force tries *all possible assignments* for the variables and checks whether the CNF is satisfied for each assignment.
- **Time Complexity:** $O(2^n)$, where n is the number of variables. This is because we try all combinations of truth values for the variables.
- **Space Complexity:** $O(n)$, because we store the assignment for each combination.
- **How it works:**
  - The algorithm loops through all possible combinations of truth values for the SAT variables.
  - For each combination, it checks if the CNF is satisfied.
  - If a valid assignment is found, it returns the solution. Otherwise, it continues until all possibilities have been exhausted.

3.2 Backtracking solver

- **Time Complexity:** Exponential, $O(2^n)$, because each variable can take two values (True/False), and we may need to explore all combinations.
- **Space Complexity:** $O(n)$, because it stores the current assignment of variables.
- **How it works:**
  - The algorithm recursively tries to assign values to each variable.
  - If a conflict occurs (i.e., the CNF is not satisfied), it backtracks and tries the next possible assignment.
  - It continues this process until a valid solution is found or all possibilities have been exhausted.

3.3 PySAT solver (using Glucose3)

- The **PySAT** solver uses an efficient SAT solver (Glucose3) to solve the CNF. Glucose3 is a state-of-the-art SAT solver known for its high performance, especially on hard SAT instances.
- **Time Complexity:** $O(2^n)$ in the worst case, but typically much faster due to the solver's optimizations
- **Space Complexity:** $O(n)$, the solver needs to store the clauses and the assignment.
- **How it works:**

- The CNF is passed to the **PySAT solver** (Glucose3), which uses advanced algorithms to solve the SAT problem.
- The solver returns a satisfying assignment if it finds one, or it returns **None** if no solution exists.

3.4 Experimentation

3.4.1 Result

- Grid size 5x5

Algorithm	Time (s)
Brute force	0.021594
Backtracking	0.00140429
PySAT	0.000170231

All solvers return same solution.
Input:

```
_, 2, _, _, _
3, _, _, 3, _
_, _, 4, _, _
_, 3, _, _, 2
_, _, _, 2, _
```

Output:

```
T, 2, T, T, G
3, G, G, 3, G
T, T, 4, T, G
G, 3, T, T, 2
G, G, G, 2, G
```

- Grid size 9x9

Algorithm	Time (s)
Brute force	12093.1
Backtracking	0.00078392
PySAT	0.000129223

All solvers return same solution

Input:

```
_, 1, 1, 1, _, _, 1, _, 1
_, 2, _, 2, 1, 1, 2, 1, 1
1, 4, _, 4, 2, _, 1, _, _
2, _, _, _, 3, 2, 2, _, _
2, _, 4, 2, 3, _, 2, _, _
1, 1, 1, 1, 4, _, 5, 2, 1
1, 1, _, 1, _, _, _, _, 3
_, 2, 2, 3, 3, 3, 5, _, _
1, 2, _, _, 1, _, 2, _, 3
```

Output:

```
G, 1, 1, 1, G, G, 1, T, 1
G, 2, T, 2, 1, 1, 2, 1, 1
1, 4, T, 4, 2, T, 1, G, G
2, T, T, T, 3, 2, 2, G, G
2, T, 4, 2, 3, T, 2, G, G
1, 1, 1, 1, 4, T, 5, 2, 1
1, 1, G, 1, T, T, T, T, 3
T, 2, 2, 3, 3, 3, 5, T, T
1, 2, T, T, 1, G, 2, T, 3
```

- Grid size 11x11

Algorithm	**Time (s)**
Brute force	> 4 hours
Backtracking	0.0019736289978027344
PySAT	0.0001690387725830078

Backtracking and PySAT return same solution

Input:

```
_, 1, 1, 1, _, 1, _, _, 1, _, _
_, 2, _, 2, 1, 2, 3, 2, 2, 1, 1
_, 2, _, 2, 1, _, 1, _, 1, _, 1
_, 2, 2, 2, 1, 2, 2, 1, 1, 1, 1
_, 2, _, 2, _, 1, _, 1, _, _, _
_, 3, _, 4, 2, 3, 3, 2, 1, 1, 1
_, 2, _, _, 3, _, _, 3, 2, _, 1
1, 2, 3, 2, 3, _, _, _, 2, 1, 1
1, _, 1, _, 1, 3, 4, 3, 2, 1, 1
2, 2, 3, 1, 2, 2, _, 3, 3, _, 1
1, _, 2, _, 2, _, 3, _, _, 2, 1
```

Output:

```
G, 1, 1, 1, G, 1, T, T, 1, G, G
G, 2, T, 2, 1, 2, 3, 2, 2, 1, 1
G, 2, T, 2, 1, T, 1, G, 1, T, 1
G, 2, 2, 2, 1, 2, 2, 1, 1, 1, 1
G, 2, T, 2, G, 1, T, 1, G, G, G
G, 3, T, 4, 2, 3, 3, 2, 1, 1, 1
G, 2, T, T, 3, T, T, 3, 2, T, 1
1, 2, 3, 2, 3, T, T, T, 2, 1, 1
1, T, 1, G, 1, 3, 4, 3, 2, 1, 1
2, 2, 3, 1, 2, 2, T, 3, 3, T, 1
1, T, 2, T, 2, T, 3, T, T, 2, 1
```

- Grid size 20x20

Algorithm	**Time (s)**
Brute force	> 4 hours
Backtracking	0.5132174491882324
PySAT	0.0004482269287109375

Backtracking and PySAT return same solution

```
_, 2, _, 1, 1, 1, 1, 2, 2, 1, _, _, _, _, _, 1, _, _, _, 1
_, 2, _, 1, _, 1, 1, _, _, 1, _, _, _, _, _, 1, 2, 3, 2, 1
1, 1, _, 1, 1, 1, 1, 2, 2, 2, 1, 1, _, _, _, _, _, _, _, _
_, _, _, _, _, _, _, 1, 1, 2, _, 1, _, _, _, _, _, 1, 1, 1
1, 1, _, _, _, _, _, 1, _, 2, 1, 2, 2, 2, 1, _, _, 2, _, 2
_, 1, _, _, _, _, 1, 2, 2, 2, 1, 2, _, _, 2, 1, _, 2, _, 2
2, 2, 1, _, _, _, 1, _, 2, 2, _, 2, 2, 3, _, 1, _, 1, 1, 1
1, _, 1, _, _, _, 1, 2, _, 2, 1, 1, _, 1, 1, 1, 1, 1, 2, 1
2, 2, 1, 1, 1, 1, _, 1, 1, 1, _, 1, 2, 2, 2, 1, 2, _, 3, _
_, 1, _, 1, _, 2, 1, _, _, _, _, 1, _, _, 2, _, 2, 1, 3, _
2, 3, 1, 2, 2, _, 1, _, _, _, _, 1, 2, 2, 2, 1, 1, _, 1, 1
_, 2, _, 1, 1, 1, 1, _, _, _, 1, 1, 1, _, _, _, _, _, _, _
2, 3, 2, 1, 1, 1, 1, _, 1, 1, 2, _, 1, _, 1, 1, 1, 1, 1, 1
1, _, 1, _, 2, _, 3, 1, 2, _, 2, 1, 1, _, 1, _, 1, 1, _, 1
1, 1, 2, 1, 4, _, 4, _, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
_, _, 1, _, 3, _, 3, 1, 1, _, _, 1, _, 1, _, _, _, _, _, _
_, _, 1, 2, 3, 3, 2, 2, 1, 1, _, 1, 1, 1, _, _, _, 1, 1, 1
_, _, _, 1, _, 2, _, 2, _, 1, _, _, _, _, _, _, _, 1, _, 1
_, _, _, 1, 2, 3, 3, 3, 2, 1, 1, 1, 1, _, _, _, 1, 2, 2, 1
_, _, _, _, 1, _, 2, _, 1, _, 1, _, 1, _, _, _, 1, _, 1, _
```

Output:

```
T, 2, G, 1, 1, 1, 1, 2, 2, 1, G, G, G, G, G, 1, T, T, T, 1
T, 2, G, 1, T, 1, 1, T, T, 1, G, G, G, G, G, 1, 2, 3, 2, 1
1, 1, G, 1, 1, 1, 1, 2, 2, 2, 1, 1, G, G, G, G, G, G, G, G
G, G, G, G, G, G, G, 1, 1, 2, T, 1, G, G, G, G, G, 1, 1, 1
1, 1, G, G, G, G, G, 1, T, 2, 1, 2, 2, 2, 1, G, G, 2, T, 2
T, 1, G, G, G, G, 1, 2, 2, 2, 1, 2, T, T, 2, 1, G, 2, T, 2
2, 2, 1, G, G, G, 1, T, 2, 2, T, 2, 2, 3, T, 1, G, 1, 1, 1
1, T, 1, G, G, G, 1, 2, T, 2, 1, 1, G, 1, 1, 1, 1, 1, 2, 1
2, 2, 1, 1, 1, 1, G, 1, 1, 1, G, 1, 2, 2, 2, 1, 2, T, 3, T
T, 1, G, 1, T, 2, 1, G, G, G, G, 1, T, T, 2, T, 2, 1, 3, T
2, 3, 1, 2, 2, T, 1, G, G, G, G, 1, 2, 2, 2, 1, 1, G, 1, 1
T, 2, T, 1, 1, 1, 1, G, G, G, 1, 1, 1, G, G, G, G, G, G, G
2, 3, 2, 1, 1, 1, 1, G, 1, 1, 2, T, 1, G, 1, 1, 1, 1, 1, 1
1, T, 1, G, 2, T, 3, 1, 2, T, 2, 1, 1, G, 1, T, 1, 1, T, 1
1, 1, 2, 1, 4, T, 4, T, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
G, G, 1, T, 3, T, 3, 1, 1, G, G, 1, T, 1, G, G, G, G, G, G
G, G, 1, 2, 3, 3, 2, 2, 1, 1, G, 1, 1, 1, G, G, G, 1, 1, 1
G, G, G, 1, T, 2, T, 2, T, 1, G, G, G, G, G, G, G, 1, T, 1
G, G, G, 1, 2, 3, 3, 3, 2, 1, 1, 1, 1, G, G, G, 1, 2, 2, 1
G, G, G, G, 1, T, 2, T, 1, G, 1, T, 1, G, G, G, 1, T, 1, G
```

3.4.2 Performance analysis

	5x5	9x9	11x11	20x20
Brute force	0.021594	12093.1	> 4 hours	> 4 hours
Backtracking	0.00140429	0.00078392	0.001973629	0.513217449
PySAT	0.000170231	0.000129223	0.000169039	0.000448227

- **Result Consistency:** All three solvers should return the **same result**.
- **Performance Differences:**
  - For **smaller grids (5x5)**, the performance difference between backtracking, brute force, and PySAT may not be significant because the problem size is manageable.
  - For **moderate grids (9x9)**, PySAT and backtracking significantly outperforms both brute force. Brute force takes a very long time to complete due to the large number of variable assignments and checks it needs to perform.
  - For **larger grids (11x11 and 20x20)**, **PySAT** will outperform backtracking and brute force. Brute force and backtracking may take a prohibitively long time due to the exponential growth in the number of variables, while PySAT can solve the problem much faster due to its optimizations.

Performance Analysis:

- **Brute force:** Brute force generates all possible assignments and checks the CNF. It has the same time complexity as backtracking, but it lacks the pruning capability, making it slower in practice. For larger grids (e.g., 11x11 or 20x20), it will become *extremely slow* due to the exponential growth in the number of variables.
- **Backtracking:** Backtracking is a *depth-first search* approach that explores the search tree recursively. It is more efficient than brute force because it can prune branches of the tree when a conflict occurs. However, its worst-case time complexity is still exponential. As the grid size increases, the number of variables grows, leading to exponential growth in the time taken (20x20 case). When it reaches a deep level and then realizes a mistake, requiring it to backtrack and retry, which also consumes additional time.
- **PySAT (Glucose3):** This solver is highly optimized and can solve large SAT problems much faster than the other two methods. It uses *CDCL* and other advanced techniques to significantly reduce the search space, making it much more efficient for large grid sizes.

3.4.3 Summary

- **Brute force** is very slow due to the lack of pruning.
- **Backtracking** is faster than brute force but still slow.
- **PySAT** is the most efficient and scalable, making it the preferred solver for larger problems.