Lab: Gem Hunter

CSC14003 - Fundamentals of Artificial Intelligence

Phan Thanh Tiến - 22120368

Table of Contents

- Table of Contents
- 1. Information
  - 1.1. Student Information
  - 1.2. Project Information
- 2. Problem Description and Logic Principles
  - 2.1. Problem Description
  - 2.2. Logic Principles
    - Proof
    - 1. "At most 2 traps" constraint:
    - 2. "At least 2 traps" constraint:
- 3. Algorithms
  - 3.1. DIMACS Format
  - 3.2. Brute-force algorithm
  - 3.3. Backtracking algorithm
  - 3.4. PySAT
- 4. Results
  - 5x5 Grid
    - Test case
    - Result
  - 11x11 Grid
    - Test case
    - Result
  - 20x20 Grid
    - Test case
    - Result
- 5. Conclusion
- 6. References

1. Information

1.1. Student Information

- **Full Name**: Phan Thanh Tiến
- **Student ID**: 22120368
- **Course**: CSC14003 - Fundamentals of Artificial Intelligence

1.2. Project Information

- Demo video link: https://youtu.be/oL49fZFDnOw
- Environment:
  - Python 3.11 or higher, with Virtual Environment (venv) enabled.
  - OS: Ubuntu 25.04
- Guide to run the project: Please refer to `README.md` file in project source code, which is in `Source` folder.
- Requirements completeness:

No	Requirement	Completeness
1	Implement a SAT solver using PySAT library	100%
2	Generate CNFs automatically from puzzle constraints	100%
3	Implement brute-force algorithm for comparison	100%
4	Implement backtracking algorithm for comparison	100%
5	Test across different puzzle sizes (5x5, 11x11, 20x20)	100%
6	Performance analysis and comparison between algorithms	100%
7	Documentation	100%

2. Problem Description and Logic Principles

2.1. Problem Description

- Given a grid of size n x n with the following symbols:
    - _: empty cell.
    - G: cell with a gem.
    - T: cell with a trap.
    - X is a positive integer: the number of traps surrounding the current cell. This cell is neither a gem cell nor a trap cell.
- Goals:
    - Find all locations of traps (T) and gems (G) in the grid, given the constraints imposed by the numbers in the cells.
    - Any cell that is not assigned as a number must be a gem or a trap.
    - The output must satisfy the condition of the numbers in the cells, meaning that the number of traps surrounding a cell must match the number in that cell.
- Assumptions:
    - If current cell contains a Trap (T), the proposition is true. In other cases, the proposition is false.
        - After this assumption, we can say that the cell is either a trap (TRUE) or not a trap (FALSE).
    - A cell containing a number X indicates that there are **exactly** X traps in the surrounding cells. The surrounding cells are the 8 adjacent cells (up, down, left, right, and the 4 diagonals).

2.2. Logic Principles

- In this problem, we are solving using Conjunctive Normal Form (CNF) and propositional logic.
- To convert the problem into CNF, we will represent each cell and its surrounding cells as boolean variables.
- As the assumption above, each cell can be represented as a boolean variable:
    - `TRUE` means the cell is a trap (T).
    - `FALSE` means the cell is not a trap (it can be a gem (G) or empty (_)).

Proof

- Now, we are going to examine a case. For simplicity but enough complexity, we will examine a case that have 2 traps.

T	**T**	?
?	**2**	?
?	?	?

- Note: ? is a unknown cell. It can be any type of cell (_, G, or X). We will not care about the exact type of the cell, for simplicity.
- Look at the center cell. It contains 2, which means that there are exactly 2 traps in the surrounding cells.
- Expressing this in CNF, we can say that the center cell is 2 if and only if the following conditions are satisfied:
    - There are no more than 2 traps in the surrounding cells.
    - There are no less than 2 traps in the surrounding cells.

1. "At most 2 traps" constraint:

- This means we cannot have 3 traps, or 4 traps, and so on, up to 8 traps among the surrounding cells.
- We can express it as: "For any group of 3 cells chosen from the 8 surrounding cells, it's impossible for all 3 of them to be traps."

To express our example case in CNF:

- Logic: If we pick any three cells, say x_i, x_j, x_k, then the statement "x_i is a trap AND x_j is a trap AND x_k is a trap" must be FALSE.
- CNF Conversion:

$$\neg(A \land B \land C) \equiv \neg A \lor \neg B \lor \neg C$$

So, for every possible combination of 3 cells $\{x_i, x_j, x_k\}$ from the 8 surrounding cells, we add the clause: $(\neg x_i \lor \neg x_j \lor \neg x_k)$ This clause means "at least one of x_i, x_j, x_k must NOT be a trap". This effectively prevents any group of 3 (or more) cells from all being traps. If we tried to make x_i, x_j, and x_k all TRUE (traps), this clause would become (FALSE $\lor$ FALSE $\lor$ FALSE), which is FALSE, violating the CNF.

2. "At least 2 traps" constraint:

- This means we must have 2 traps, or 3 traps, ..., or 8 traps.
- A common way to express "at least k" is to say: "Out of N surrounding cells, you cannot have more than N-k non-traps."

In our case, $N = 8$ (surrounding cells) and $k = 2$ (traps). So, we cannot have more than $8 - 2 = 6$ non-traps. This is equivalent to saying: "If you pick any 7 cells (which is N-k+1 = 8-2+1 = 7), at least one of them must be a trap."

- Logic: If we pick any group of 7 cells from the 8 surrounding cells, say $x_a, x_b, x_c, x_d, x_e, x_f, x_g$, then the statement "$x_a$ is NOT a trap AND x_b is NOT a trap ... AND x_g is NOT a trap" must be FALSE.
- CNF Conversion:

$$\neg(\neg A \land \neg B \land ... \land \neg G) \equiv A \lor B \lor ... \lor G.$$

  So, for every possible combination of $N - k + 1 = 8 - 2 + 1 = 7$ cells $\{x_a, ..., x_g\}$ from the 8 surrounding cells, we add the clause: $(x_a \lor x_b \lor x_c \lor x_d \lor x_e \lor x_f \lor x_g)$ This clause means "at least one of $x_a, ..., x_g$ must be a trap". This prevents having too many non-traps (which would imply too few traps).

3. Algorithms

3.1. DIMACS Format

- To make the solver read data easier, we will use the DIMACS format to represent our boolean formulas.
- DIMACS syntax:
  - The first line contains `p cnf n m`, with `n` is the number of variables, `m` is the number of clauses.
  - The next `m` lines contain the clauses, each clause is a list of integers that:
    - Positive integers represent the variable.
    - Negative integers represent the negation of the variable.
    - Each clause ends with a `0`.
  - Multiple literals in a clause represent a logical OR (∨)
  - Multiple clauses represent a logical AND (∧)
- For example:

$$(\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3)$$

- The above formula can be represented in DIMACS format as follows:

```
p cnf 3 2
-1 2 0
-2 3 0
```

- Explanation:
  - `p cnf 3 2` means 3 variables and 2 clauses
  - `-1 2 0` represents the clause $(\neg x_1 \lor x_2)$
  - `-2 3 0` represents the clause $(\neg x_2 \lor x_3)$

3.2. Brute-force algorithm

For this algorithm, we will try all set of value for the CNF formula. If result is `TRUE`, that means that the current set of value is a solution. If result is `FALSE`, we will try another set of value.

The worst case is that we will try all possible set of value, which is 2^n where n is the number of variables. This algorithm is infeasible for large grids. In my program, I have limited at most 100M loops to try the set of values. Both 11x11 and 20x20 grids cannot be solved with this limitation. (Actual runtime is more than 30 minutes for trying 100M loops.)

- Time complexity: $O(2^n)$ where n is the number of variables.
- Space complexity: $O(n)$ where n is the number of variables.

3.3. Backtracking algorithm

The backtracking algorithm is a more efficient way to solve the problem by exploring the search space systematically. It tries to build a solution incrementally, abandoning paths that cannot lead to a valid solution.

I am actually using Depth-First Search (DFS) to implement the backtracking algorithm. The algorithm works as follows:

1. Start with an empty assignment of variables.
2. Select a variable and assign it a value (TRUE or FALSE).
3. Check if the current assignment is consistent with the constraints of the problem.
4. If consistent, recursively proceed to the next variable.
5. If all variables are assigned and the assignment is valid, return the solution.
6. If inconsistent, backtrack by removing the last assignment and trying the next value for that variable. -> This helps pruning search space for cases that unsolveable.

- Time complexity: $O(n!)$ where n is the number of variables. This is because we may need to try all permutations of variable assignments.
- Space complexity: $O(n)$ where n is the number of variables. This is due to the recursive nature of the algorithm and the storage of variable assignments.

3.4. PySAT

The PySAT library provides a powerful interface for working with SAT solvers. It allows us to create CNF formulas, add clauses, and solve them using various SAT solvers.

- In this implementationm, I am using Glucose4 as the SAT solver.

- The Glucose 4 solver used by PySAT primarily implements the CDCL (Conflict-Driven Clause Learning) algorithm:

- Uses VSIDS (Variable State Independent Decaying Sum) to select the next variable to assign.
- Efficiently propagates implications of assignments using a watched literal scheme.
- Conflict Analysis: When contradictions are detected, analyzes the conflict to:
    - Learn new clauses that prevent similar conflicts
    - Determine the appropriate backtracking level
    - Prune large portions of the search space
- Restart Strategies: Periodically restarts the search to escape from unproductive areas of the search space.
- Clause Deletion: Manages learned clauses, keeping useful ones and discarding others to control memory usage.
- Time complexity: SAT is NP-complete, so the worst-case time complexity is exponential in the number of variables. However, practical performance is often much better due to the efficiency of modern SAT solvers.
- Space complexity: Depends on the number of clauses and variables, but typically $O(n + m)$ where n is the number of variables and m is the number of clauses.

4. Results

5x5 Grid

Test case

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

Result

Algorithm	Time (seconds)	Compared to SAT
SAT	0.000145	-
Brute-force	0.000910	6.28x slower
Backtracking	0.000079	0.55x faster

11x11 Grid

Test case

```
11 11
_,_,1,_,_,1,_,1,_,_,_
2,3,2,2,2,3,2,1,1,1,1
1,3,_,3,_,_,2,_,2,_,2
1,_,_,5,4,_,3,2,3,_,3
1,4,_,_,4,5,_,4,_,6,_
1,4,_,6,_,_,_,6,_,_,_
1,_,_,6,_,_,6,_,_,5,_
2,3,4,_,_,4,_,_,4,3,2
1,_,3,4,3,4,4,5,4,_,1
2,4,_,3,_,2,_,_,_,3,1
1,_,_,3,1,2,2,4,_,2,_
```

Result

Algorithm	Time (seconds)	Compared to SAT
SAT	0.000151	-
Brute-force	N/A	Much slower. Cannot be solved in feasible time
Backtracking	0.001288	8.54x slower

20x20 Grid

Test case

```
20 20
_,2,1,_,3,2,1,_,_,_,_,1,_,_,4,_,_,_,_,1
_,3,3,4,_,_,1,_,_,_,1,2,5,_,_,5,_,5,2,1
2,3,_,_,3,3,3,2,1,_,1,_,5,_,4,3,_,3,2,2
_,2,3,3,2,1,_,_,2,_,1,3,_,_,2,1,1,2,_,_
2,2,1,_,1,2,4,_,3,2,2,3,_,4,3,3,2,2,2,2
_,3,2,2,2,2,_,2,2,_,_,3,1,2,_,_,_,1,1,1
_,3,_,1,1,_,3,3,3,4,_,2,_,2,4,_,3,1,2,_
1,3,2,2,1,1,2,_,_,3,2,1,_,1,_,3,2,1,2,_
_,1,_,1,_,1,3,4,4,_,2,2,2,2,2,4,_,3,2,1
_,1,2,2,2,2,_,_,3,1,2,_,_,2,1,_,_,_,1,_
_,1,2,_,3,_,5,_,2,_,1,3,_,3,2,3,5,5,3,1
_,1,_,3,_,3,_,2,1,_,_,1,2,_,1,1,_,_,_,1
_,1,2,3,2,2,1,1,_,_,_,1,2,3,2,2,2,4,3,2
1,1,2,_,2,1,1,2,1,1,_,1,_,3,_,3,1,2,_,2
_,1,2,_,2,2,_,3,_,1,_,1,1,3,_,_,1,2,_,2
1,1,1,2,2,3,_,3,1,1,_,_,_,1,2,2,2,2,2,1
1,2,1,3,_,4,3,3,2,1,1,1,1,1,1,1,2,_,1,_
_,2,_,3,_,3,_,_,3,_,2,1,_,2,3,_,3,1,1,_
1,2,2,3,3,4,5,4,4,_,3,2,3,_,4,_,4,1,1,_
_,_,1,_,2,_,_,_,2,1,2,_,2,1,3,_,3,_,1,_
```

Result

Algorithm	**Time (seconds)**	**Compared to SAT**
SAT	0.000357	-
Brute-force	N/A	Much slower. Cannot be solved in feasible time
Backtracking	0.012609	35.34x slower

5. Conclusion

- SAT algorithm is the most efficient algorithm for this kind of problem.
- Brute-force algorithm is not feasible for large grids, as it takes too long to try all possible combinations.
- Backtracking algorithm is more efficient than brute-force, but still slower than SAT solver. Backtracking algogrithm can simplify the search space by pruning unsolvable paths, but it still cannot compete with the efficiency of modern SAT solvers like Glucose4.

6. References

[1] Antonio Morgado Alexey Ignatiev Joao Marques-Silva. Boolean formula manipulation (pysat.formula). Accessed: 12 May 2025. url: https://pysathq.github.io/docs/html/api/formula.html