

VNUHCM-UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

CSC14003 – INTRODUCTION TO ARTIFICIAL INTELLIGENCE

---

## Lab 02: Gem Hunter

---

**Lecturer**

Ms. Nguyễn Ngọc Thảo  
Mr. Nguyễn Trần Duy Minh

**Class**

**23CLC08**

**Students**

23127226 – Nguyễn Đăng Minh



April 7th, 2025

# Contents

<b>1</b>	<b>Project Structure and Execution</b>	<b>2</b>
<b>2</b>	<b>Degree Of Completion Level</b>	<b>3</b>
<b>3</b>	<b>Verifying The Results</b>	<b>3</b>
<b>4</b>	<b>Generating Conjunctive-Normal-Form Sentences</b>	<b>4</b>
4.1	Upper-bound sentence . . . . .	5
4.2	Lower-bound sentence . . . . .	6
<b>5</b>	<b>Gem-Hunter Solving Algorithms</b>	<b>6</b>
5.1	Pysat Library Approach . . . . .	6
5.2	Backtracking Approach . . . . .	7
5.3	Brute Force Approach . . . . .	8
<b>6</b>	<b>Experiments</b>	<b>10</b>
6.1	Time Comparison . . . . .	10
6.2	Result Verifying . . . . .	11

# 1 Project Structure and Execution

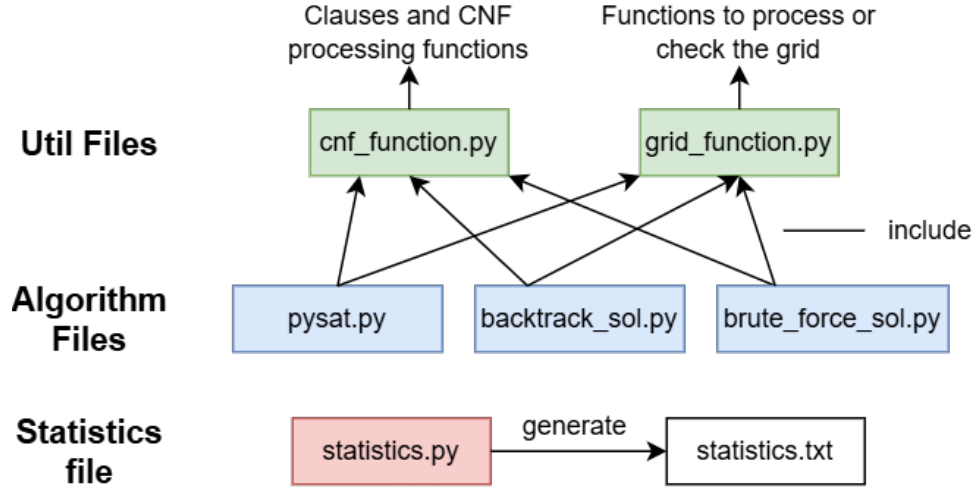


Figure 1: The code organization

The project organization is shown in Figure 1 above. There are three types of file category:

1. *Util Files* include functions of clauses, CNF, and grid that are used many times.
2. *Algorithm Files* will use different algorithms to solve the problem. All of them will include the *Util Files* to use the corresponding functions.
3. *Statistics File* is used to print the running information of all algorithms along with the test cases. It is mainly used to experiment and compare between algorithms.

## 2 Degree Of Completion Level

Assignment	Subtask	Progress (%)
CNF		
	Correct logical principles of CNFs	100%
	Generate CNFs automatically	100%
	Remove duplicate clauses	100%
PySAT		
	Use PySAT library to solve CNFs correctly	100%
	Get the speed	100%
Backtrack		
	Use Backtrack with optimization to solve CNFs correctly	100%
	Get the speed	100%
Brute Force		
	Use Brute Force to solve CNFs correctly	100%
	Get the speed	100%
Algorithms comparison	Comparison in the report	100%
Test cases		
	Have various-size test cases	100%
	Have 6 test cases	100%
Video	Demo Running Process	100%
Report	Project Document	100%

## 3 Verifying The Results

In the process of coding, there must be a function to test if the result grid is valid in the Gem Hunter problem. The method of checking is simple as it will traverse through each number cell and count the number of traps around it. This ensures the confidence when comparing the algorithms as their results are believed to be accurate.

If it is false, then the result grid returned by some arbitrary algorithms is wrong. Below is the pseudo-code for validating the grid. The `trapCount()` function will calculate the number of traps in surrounding cells.

---

```

1: function CHECK_VALID_GRID(result_grid)
2:   for  $i = 0$  to num_row do
3:     for  $j = 0$  to num_col do
4:       if isNumberCell( $i, j$ ) and trapCount( $i, j$ )  $\neq$  result_grid[ $i$ ][ $j$ ] then
5:         return False
6:       end if
7:     end for
8:   end for
9:   return True
10: end function

```

---

## 4 Generating Conjunctive-Normal-Form Sentences

How the Conjunctive Normal Form sentences are created to solve the Gem Hunter problem?

*First, I will denote that the value True of a variable means that there is a trap at the cell corresponding to that variable. Similarly, False value indicates the Gem.*

Now, consider the Wumpus World problem where the agent may have sensors on each square, for example, a Breeze at square  $(x, y)$  indicates that there is a pit surrounding. In the Gem Hunter, I will use the number cell: number of traps surrounding it, as the sensors for the agent.

The easiest inference for the agent is that the number cell is neither the trap nor the gem cell. As the gem cells are not as important as the trap cell, the value of assignment for the number cell is False. For example, in Figure 2 below, the  $X2$  cell has value 2, so the CNF sentence added to the knowledge base is:  $\neg X2$

Cell X2



Figure 2: Number-cell case

How about the unknown cells? Let us consider a number cell with value  $x$  which indicates that there are exactly  $\#x$  traps surrounding it in Figure 3 below.

Exactly  $x$  traps

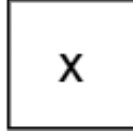


Figure 3: Exactly  $x$  traps in the cell

We can write sentences such as:

$\exists$  (combination of  $x$  cells out of 8) such that  $(x \text{ cells are traps}) \wedge (8 - x \text{ cells are not traps})$

This is true, however, it is very hard (or I have not found a way) to transform this sentence to Conjunctive Normal Form. Thus, I will divide the "Exactly  $x$  traps" sentence into two "weaker" sentences like in Figure 4 below. Let us call the "At most" sentence be the **upper-bound sentence** and the other one **lower-bound sentence**.

(At least  $x$  traps)  $\wedge$  (At most  $x$  traps)

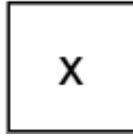


Figure 4: Two "weaker" sentences connected by **AND** logic

## 4.1 Upper-bound sentence

Consider a situation in Figure 5 below.

X1	X2
2	X4
X5	X6

Figure 5: An example

The "At most 2 traps" sentence is actually the negation of "At least 3 traps". The "At least 3 traps" sentence is expressed as below:

$(X1 \wedge X2 \wedge X4) \vee (X1 \wedge X2 \wedge X5) \vee (X1 \wedge X2 \wedge X6) \vee \dots \vee (X4 \wedge X5 \wedge X6)$

It is basically all the combinations of 3 cells out of 6 neighbor cells connected via the  $\vee$  symbol. After negating the sentence, we get the final upper-bound sentence of the CNF:

$$(\neg X1 \vee \neg X2 \vee \neg X4) \wedge (\neg X1 \vee \neg X2 \vee \neg X5) \wedge (\neg X1 \vee \neg X2 \vee \neg X6) \wedge \dots \wedge (\neg X4 \vee \neg X5 \vee \neg X6)$$

In conclusion, the "At most  $k$  traps" sentence can be transformed to CNF by negating the "At least  $k + 1$  traps" sentence.

$$\text{"At most } k \text{ traps"} \leftrightarrow \neg \text{"At least } k + 1 \text{ traps"}.$$


---

## 4.2 Lower-bound sentence

Consider the same example in Figure 5. If we perform the same operation as Upper-bound sentence: "At least  $k$  traps"  $\leftrightarrow$   $\neg$ "At most  $k - 1$  traps", it would not yield the CNF because the Upper-bound is already a CNF (proved above). The main problem is because of the **NOT** logic. Luckily, we can get rid of the NOT logic by slightly changing the definition:

$$\text{"At least } k \text{ traps"} \leftrightarrow \text{"At most } n - k \text{ gems"} \text{ (} n \text{ is the number of } \mathbf{valid} \text{ neighbor cells)}$$

Because gem is the "negation" of trap (False and True, respectively), so we can take the same Upper-bound sentence and change only the sign of the variables. For example, "At most 3 gems" will be expressed as:

$$(X1 \vee X2 \vee X4) \wedge (X1 \vee X2 \vee X5) \wedge (X1 \vee X2 \vee X6) \wedge \dots \wedge (X4 \vee X5 \vee X6)$$

In conclusion, the "At least  $k$  traps" will be written as "At most  $n - k$  gems" ( $n$  is the number of **valid** neighbor cells). Since "At most  $n - k$  gems"  $\leftrightarrow$   $\neg$ "At least  $n - k + 1$  gems", then "At least  $k$  traps"  $\leftrightarrow$   $\neg$ "At least  $n - k + 1$  gems".

# 5 Gem-Hunter Solving Algorithms

## 5.1 Pysat Library Approach

Once we have generated the CNF sentence, we need to add all its clauses to the CNF() sentence of PySAT library [2]. Its format is:  $[-1, 2], [3, -4]$  indicates  $(\neg X1 \vee X2) \wedge (X3 \vee \neg X4)$ . The negative sign is similar to the  $\neg$ .

Initialize the PySAT solver with the CNF as the parameter. Then call the solve() function and get the model.

The **PySAT** solver will solve the CNF clauses incredibly fast. Even for more than 20000 clauses, it took only 0.0000ms to solve.

Below is the code to solve the CNF by **PySAT**:

```
solver = Solver(bootstrap_with = cnf)
if solver.solve():
```

```
# Model format will contain all variables and their sign
model = solver.get_model()
else:
    print("Unsatisfied")
```

## 5.2 Backtracking Approach

The backtracking algorithm will recursively try to assign value for each variables (only variables that stand for unknown cells). For each unknown variables which stand for unknown cells, first try to assign the True value, and if not yet succeed, backtrack again and attempt the False value. The test cases are made so that there always exists a solution.

However, a pure backtrack is similar to generating a whole truth table, which would yield the time complexity of  $O(2^n)$ . Thus, optimization is necessary. The optimization idea is alike to the DPLL algorithm [1], which applies early termination, pure symbol, and unit clauses heuristics. After applying the necessary heuristics, the backtracking algorithm works substantially fast as it could solve for a 30x30 grid (21094 clauses) in only 7 seconds.

Below is the pseudo-code for the backtracking algorithm with optimization:



---

**Algorithm 1** Backtracking algorithm

---

```
1: function FIND_UNIT_CLAUSE(clauses, model) returns unit clause or null
2:   for clause in clauses do
3:     if No_True_Assignment(clause) and #unassigned(clause) == 1 then
4:       return unassigned literal, sign
5:     end if
6:   end for
7:   return null
8: end function
9:
10: function FIND_PURE_SYMBOL(clauses, model) returns pure literal or null
11:   for literal in clauses do
12:     if Only_One_Sign(literal) in clauses then
13:       return literal, sign
14:     end if
15:   end for
16:   return null
17: end function
18:
19: function BACKTRACK(variables, clauses, model) returns True or False
20:   if exist false clause in clauses then return False ▷ Early termination
21:   if all clauses in clauses true then
22:     print model
23:     return True
24:   end if
25:   unitP, value ← Find_Unit_Clause(clauses, model) ▷ Unit-clause heuristic
26:   if unitP ≠ null then
27:     return BACKTRACK(variables – unitP, clauses, model ∪ {unitP = value})
28:   end if
29:   pureP, value ← Find_Pure_Symbol(clauses, model) ▷ Pure-symbol heuristic
30:   if pureP ≠ null then
31:     return BACKTRACK(variables – pureP, clauses, model ∪ {pureP = value})
32:   end if
33: end function
```

---

### 5.3 Brute Force Approach

Brute Force approach will generate all  $O(2^n)$  possible bit-mask with  $n$  is the number of variables standing for unknown cells. Consider a simple situation below in Figure 6.

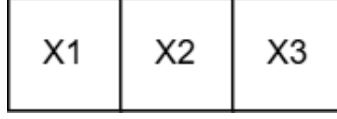


Figure 6: Brute Force example

Let us denote bit 1 in the bitmask be the trap, and 0 be the gem. For the example, we would have  $2^3 = 8$  possible cases.

$[\{0, 0, 0\}, \{0, 0, 1\}, \{0, 1, 0\}, \{0, 1, 1\}, \{1, 0, 0\}, \{1, 0, 1\}, \{1, 1, 0\}, \{1, 1, 1\}]$

For each mask that we generate, simply check if it satisfies the CNF. Here is the pseudo-code for the brute-force approach:

---

**Algorithm 2** Brute Force

---

```

1: function BRUTE_FORCE(clauses, model) returns mask or -1
2:   for mask  $\in (2^n \text{ subsets})$  do
3:     if SATISFIED(clauses, mask) then
4:       return mask
5:     end if
6:   end for
7:   return -1
8: end function

```

---

However, in the project, I performed the "Iterative Brute Force", which iterates over the number of traps and check only masks that has the bit count of 1 equals number of traps. The pseudo-code for "Iterative Brute Force" is shown below. The NEXT\_MASK function will get the new larger mask which have the same number of bit 1 count of current mask (if possible).

---

**Algorithm 3** Iterative Brute Force (used in project)

---

```

1: function ITERATIVE_BRUTE_FORCE(clauses, model) returns mask or -1
2:   for trap_cnt = 0  $\rightarrow$  num_unknown_cells do
3:     mask  $\leftarrow (1 \ll \text{trap\_cnt})$   $\triangleright$  smallest mask of bit_count = trap_cnt
4:     while mask <  $(1 \ll \text{num\_unknown\_cells})$  do
5:       if SATISFIED(clauses, mask) then
6:         return mask
7:       end if
8:       mask  $\leftarrow$  NEXT_MASK(mask)  $\triangleright$  next larger mask with same bit_count
9:     end while
10:  end for
11:  return -1
12: end function

```

---

The overall time complexity of two algorithms are  $O(2^n)$ .

## 6 Experiments

### 6.1 Time Comparison

Instead of running each test cases, we can execute the `statistics.py` file and read all the statistics in the `statistics.txt` file. The statistics table is shown in Figure ?? after executing the file.

Grid size	# Clauses	# Empty cells	Algorithm	Time
5x5	496	11	pysat	0.0000ms
5x5	496	11	backtrack	1.2541ms
5x5	496	11	bruteforce	0.0000ms
6x6	621	24	pysat	0.0000ms
6x6	621	24	backtrack	2.5296ms
6x6	621	24	bruteforce	20.0171ms
10x10	1547	67	pysat	0.0000ms
10x10	1547	67	backtrack	48.8117ms
10x10	1547	67	bruteforce	Long
15x15	5695	129	pysat	0.0000ms
15x15	5695	129	backtrack	300.1463ms
15x15	5695	129	bruteforce	Long
20x20	10453	235	pysat	6.2177ms
20x20	10453	235	backtrack	1660.1064ms
20x20	10453	235	bruteforce	Long
30x30	21094	528	pysat	0.0000ms
30x30	21094	528	backtrack	6849.3955ms
30x30	21094	528	bruteforce	Long

Figure 7: Statistics Table

Based on the statistics table above:

- **PySAT:** in every cases PySAT is always the fastest one. The overall time in milliseconds for the Gem-Hunter solver using PySAT is 0.0000ms, even for the 30x30 grid (21094 clauses), which is incredible.
- **Backtrack:** The Backtracking algorithm with optimization also performs well as the search space is effectively pruned. For the largest test case, backtrack only took nearly 7 seconds to solve.

- **Iterative Brute Force:** Iterative Brute Force approach is the longest one and there is no optimization applied. As mentioned in section 4.3, the time complexity for Brute-Force algorithm is  $O(2^n)$ . Moreover, I have tested for many other test cases and found out that Iterative Brute Force seems to be slightly faster than Brute Force approach. In conclusion, Brute Force algorithms can only run in reasonable time if the number of unknown cells are less than 25.

## 6.2 Result Verifying

Let us consider the result for the 20x20 test cases (test case 5).

1	-	-	-	2	2	2	1	1	1	2	1	1	1	-	2	1	-	2	2
-	-	-	4	3	-	-	2	2	-	-	-	1	-	-	-	-	3	-	-
-	4	2	-	-	-	-	3	-	3	-	-	3	-	3	-	4	-	4	-
-	2	-	-	-	2	-	-	-	2	-	3	-	-	4	-	1	4	-	3
2	3	3	-	-	2	-	1	-	-	-	-	5	-	-	2	1	4	-	4
-	-	-	2	-	1	2	2	2	3	-	-	-	-	-	-	-	4	-	-
3	4	3	-	-	-	-	-	-	3	-	3	1	1	1	-	2	-	-	-
-	-	1	-	-	3	3	2	2	2	-	-	1	1	-	-	1	1	2	2
-	3	2	-	-	-	3	2	-	1	-	2	-	3	2	3	2	1	-	-
-	-	-	2	-	-	4	-	-	2	-	3	-	4	-	2	-	-	2	-
-	-	-	5	-	-	-	4	-	-	-	-	4	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-	2	1	-	1
1	2	4	-	-	3	4	-	3	2	4	-	4	-	-	-	-	-	-	-
2	-	-	1	3	-	-	-	-	-	-	-	-	-	-	3	-	2	-	-
-	2	1	-	2	-	-	4	4	5	-	-	-	-	4	-	3	-	2	1
1	1	-	1	3	4	-	2	-	-	5	-	3	2	-	-	-	3	-	-
-	-	-	-	-	3	4	-	6	-	-	2	-	2	-	-	-	3	-	-
-	-	-	1	2	2	3	-	-	-	-	2	1	1	-	-	4	-	-	-
-	-	-	-	2	1	2	-	-	5	4	4	-	-	2	-	4	3	-	-
-	-	1	-	-	-	1	2	-	-	2	-	3	-	3	-	-	-	2	-

Figure 8: Test case 5

Below is the result grid by PySAT and Backtrack algorithm:

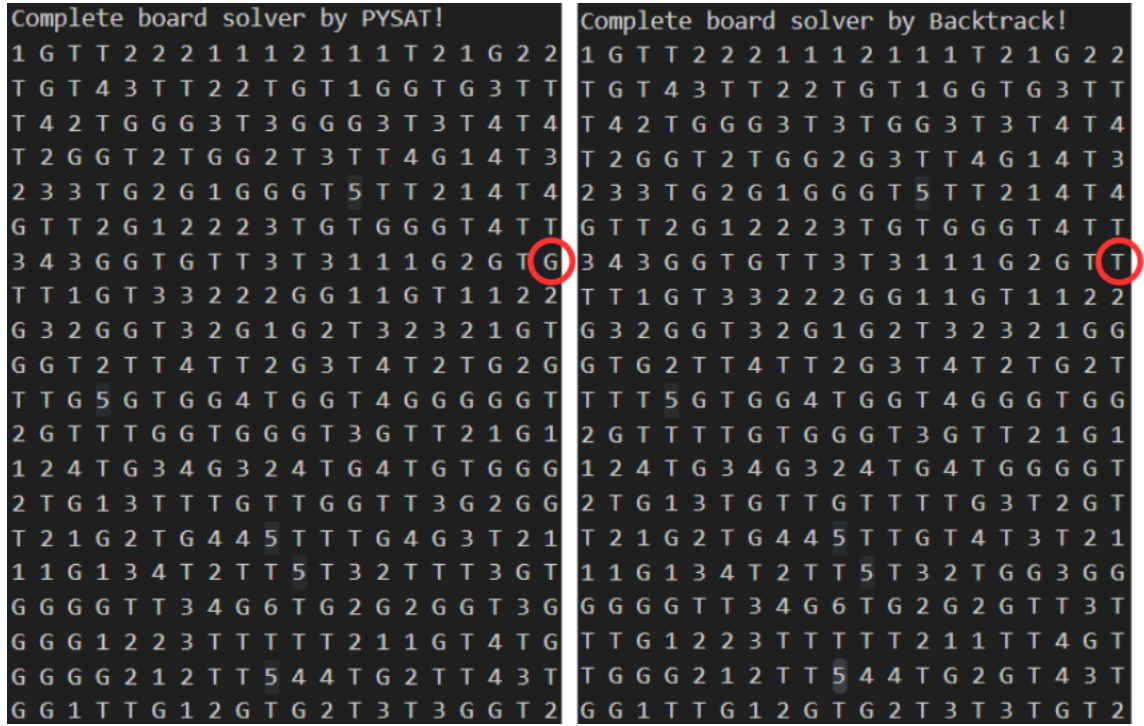


Figure 9: Result of PySAT and Backtrack

Since the result grid generated by Backtrack algorithm has been verified by the `CHECK_VALID_GRID` function in section 2, it is ensured to be accurate. Besides, if we look closely, the result of back-track is slightly different from PySAT, which is marked by the red circle.

## References

- [1] Wikipedia. *DPLL algorithm*. URL: [https://en.wikipedia.org/wiki/DPLL\\_algorithm](https://en.wikipedia.org/wiki/DPLL_algorithm).
- [2] Wikipedia. *PySAT Documentation*. URL: <https://pysathq.github.io/docs/html/api/solvers.html>.