

VNUHCM-UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

CSC14003 – INTRODUCTION TO ARTIFICIAL INTELLIGENCE

LAB: Gem hunter

Lecturer

Ms. Nguyễn Ngọc Thảo
Mr. Nguyễn Thanh Tình

Class

23CLC08

Students

23127255 – Nguyễn Thọ Tài



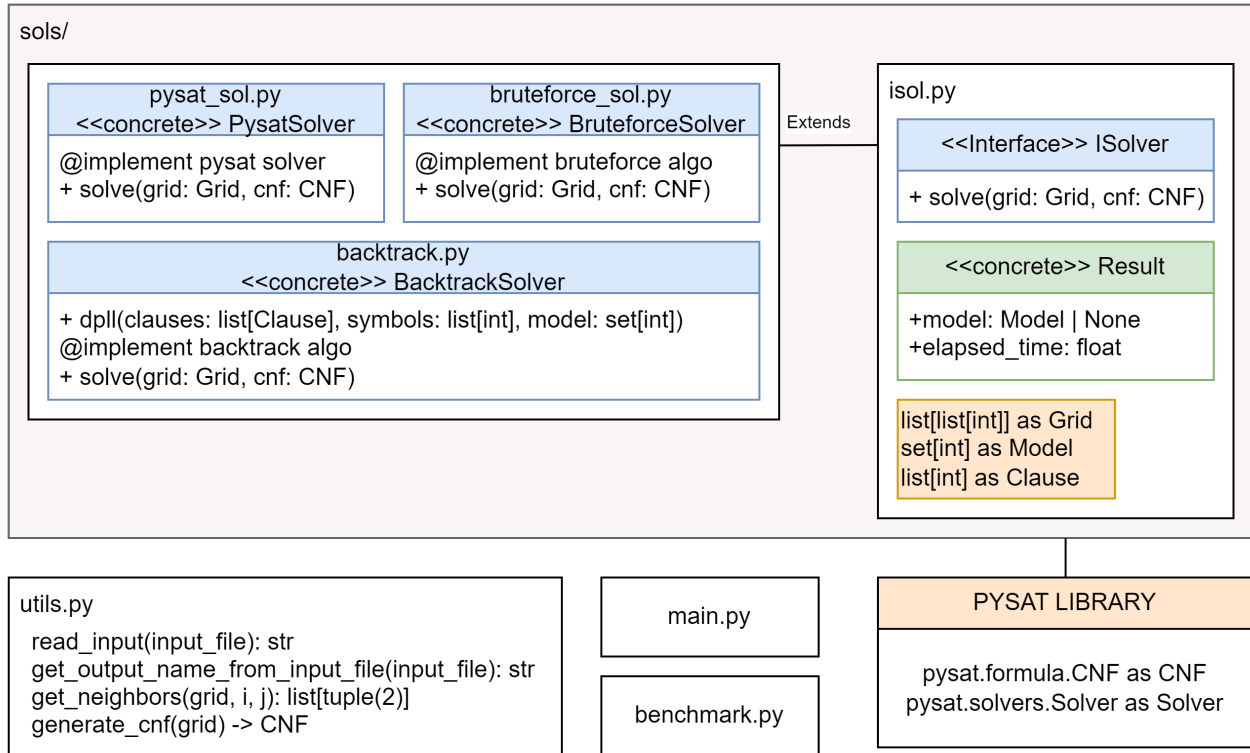
April 7th, 2025

Contents

1	General	2
1.1	Lab structure	2
1.2	Lab Completion	3
2	Generating CNF Sentences	3
2.1	At least x trap surrounding	3
2.2	At most x trap surrounding	3
2.3	Remove duplicate clause	4
3	Solver Algorithms	4
3.1	pySAT Approach	4
3.2	Backtrack Approach	4
3.3	Brute-force Approach	5
4	Experiment	6
4.1	Benchmark result	6
4.2	Cells out of scope	7
4.3	Different results between solvers	7

1 General

1.1 Lab structure



The image above is the structure of the source code `src/`, each files can be explained as the following:

1. `sols/isol.py`: contain the interface `ISolver` and `Result` structure. The `ISolver.solve()` return either `Result` or `None`. The `Result` store the satisfied model and the elapsed time to solve the `CNF`.
2. `sols/*_sol.py`: the implementation of said solvers for this gem hunter LAB.
3. `utils.py`: contain all the utility functions that was commonly use across the source code such as `generate_cnf()`, `read_input()`, etc...
4. `main.py`: run the all the inputted input files dump the result to the screen and output files.
5. `benchmark.py`: the advanced version of `main.py` which has a built-in timeout system that stop the solver once the solving take too long to finish. This also perform a summary of all the results.

1.2 Lab Completion

Assignment	Subtask	Progress (%)
CNF	Generate Correct CNFs Automatic	100%
	Optimize CNFs to have fewer clauses	100%
	Remove duplicate clauses	100%
PySAT	Use PySAT library to solve CNFs	100%
	Benchmarking	100%
Backtrack	Use Backtrack (DPLL) with optimization to solve CNFs	100%
	Benchmarking	100%
Brute Force	Use Brute Force to solve CNFs	100%
	Benchmarking	100%
Solvers comparison	Print out the summary of the comparison	100%
	Comparison in the report	100%
1*Test cases	Have more than 3 test cases and they are varied in size.	100%
1*Video	Demo Running Process	100%
1*Report	Project Documentation	100%

Table 1: Progress Table

2 Generating CNF Sentences

For every numbered cell x , there are exactly x traps surrounding that cell \equiv there are at least x traps surrounding and at most x traps surrounding.

2.1 At least x trap surrounding

Supposing there are n valid cells surrounding the x cell, there are at least x trap surrounding \equiv at most $n - x$ non-traps cells \equiv not(at least $n - x + 1$ non-traps)
not(at least $n - x + 1$ non-traps): $(x_1 \wedge x_2 \wedge \dots \wedge x_{(n-x+1)}) \equiv (x_1 \vee x_2 \vee \dots \vee x_{(n-x+1)})$
 $\rightarrow \bigwedge_{\text{combinations of } n-x+1 \text{ non-traps}} \bigvee(\text{combination})$

2.2 At most x trap surrounding

At most x trap surrounding \equiv not(at least $x + 1$ traps surrounding)
 $\rightarrow \bigwedge_{\text{combinations of } x+1 \text{ traps}} \bigvee(\text{combination})$

2.3 Remove duplicate clause

In my implementation, I used `set()` data structure in python which is a hash table... as I add new clause to `clause_set`; it will only get added one due to both the duplicate and the existing one have the exact same hash; also doing so give me $O(1)$ advantage when check if a literal/symbols exist in the `clause_set`.

Tho, there was an issue where this two identical clauses `[1, 2, 3]` and `[1, 3, 2]` both get added separately. This was later patched as now the added tuple is a sorted one thus the order was always the same.

3 Solver Algorithms

3.1 pySAT Approach

Since the CNF clauses are provided by the pySAT library, we simply pass them directly to the solver via "bootstrap_with" parameter, then we call to the `solve()` method of the solver which handles the rest.[\[3\]](#)

```
solver = Solver(bootstrap_with=cnf)
if solver.solve():
    model = solver.get_model()
else:
    # cannot be solved
```

3.2 Backtrack Approach

The backtracking solver in my lab implements the DPLL (Davis–Putnam–Logemann–Loveland) algorithm for SAT solving, with three optimizations: early termination (returns the model when all clauses are satisfied or unsatisfiable), pure symbol heuristic (assigns values to variables appearing in one only sign), and unit clause heuristic (propagates values for literals that are the only unassigned literals in any clause). These optimizations reduce the search space and enhance algorithm efficiency.[\[2\]](#)[\[1\]](#)

Algorithm 1 DPLL - Davis–Putnam–Logemann–Loveland

```
1: FIND_PURE_SYMBOL(clauses, model)
2:   pure_literals  $\leftarrow \{\}$ 
3:   for each literal in each clause in clauses then
4:     if literal is assigned in model then continue
5:     if -literal in pure_literals then pure_literals[-literal]  $\leftarrow$  false
6:     else if literal not in pure_literals then pure_literals[literal]  $\leftarrow$  true
7:   return FIRST_TRUE(pure_literals)
8:
9: FIND_UNIT_CLAUSE(clauses, model)
10: for each clause in clauses then
11:   if any literal in clause is assigned in model then continue
12:   if clause only has one literal is not assigned in model then return that literal
13: return 0
14:
15: DPLL(clauses, symbols, model)
16:   if all clause in clauses is true in model then return model
17:   if any clause in clauses is false in model then return null
18:   P  $\leftarrow$  FIND_PURE_SYMBOL(clauses, model)
19:   if p is not 0 then return DPLL(clauses, symbols - P, model  $\cup$  {P})  $\triangleright$  0 mean none,
   -3 is 3 is neg, 4 mean 4 is pos; the number is mapping via row * col_size + col + 1
20:   P  $\leftarrow$  FIND_UNIT_CLAUSE(clauses, model)
21:   if p is not 0 then return DPLL(clauses, symbols - P, model  $\cup$  {P})
22:   P  $\leftarrow$  FIRST(symbol), symbol  $\leftarrow$  symbol - {P}
23:   return DPLL(clauses, symbol, model  $\cup$  {P}) or DPLL(clauses, symbol, model  $\cup$  {-P})
```

3.3 Brute-force Approach

This algorithm perform brute-force to find the combination where the model is satisfied. In my implementation, it first cached up all neighbor cells which is to save the algorithm from repetitively calling to "GET_NEIGHBORS()" function.

The core of the algorithm is the model satisfaction check, where it checks every numbered cells that if there are exactly that number of trap surrounding that cell. It then performs a brute-force search, testing all combinations of unknown cells (trap locations) from the maximum number of traps to zero. This exhaustive search guarantees that if a solution exists, it will be found.

The time complexity of the algorithm is $O(2^n)$, where *n* is the number of unknown cells, making it really slow when perform with large grids. While guaranteed to find a solution, the brute-force nature of the algorithm makes it impractical for larger problems, and more efficient algorithms, such as the two previous approaches are typically preferred in such cases due to their ability to reduce the search space.

Algorithm 2 Bruteforce Solution

```
1: function solve(grid, cnf)
2:   rows, cols  $\leftarrow$  dimensions of grid
3:   unknown_cells  $\leftarrow \{\}$ 
4:   neighbors_cache  $\leftarrow \{\}$ 
5:   for each (i, j) in grid do
6:     neighbors_cache[(i, j)]  $\leftarrow$  get_neighbors(grid, i, j)
7:   for each (i, j) in grid do
8:     if cell at (i, j) is number then
9:       for each neighbor (r, c) in neighbors_cache[(i, j)] do
10:        if cell at (i, j) is not number then
11:          unknown_cells  $\leftarrow$  unknown_cells  $\cup \{(r, c)\}$ 
12:   for trap_count from len(unknown_cells) down to 0 do
13:     for each combination traps in unknown_cells with trap_count elements do
14:       if MODEL_IS_SATISFIED(traps) then
15:         mapped_model  $\leftarrow [r * \text{cols} + c + 1 \text{ for } (r, c) \text{ in traps}]$ 
16:         return mapped_model
17:   return null
```

4 Experiment

4.1 Benchmark result

You can obtain the results by running either *benchmark.py* or *main.py*. Both perform benchmarking, but only *benchmark.py* includes a timeout mechanism to detect excessively long runs (time out) and perform the summary of the data.

Note that performance may slightly vary between the two due to Python’s runtime environment (e.g., GIL), which can affect comparison outcomes.

Input File	Clauses	Empty Cells	PySAT (ms)	Backtrack (ms)	Brute-force
input_1.txt	67	11 (4x4)	0.0758	0.9174	2.0819
input_2.txt	298	38 (7x7)	0.2286	59.2687	Time Out
input_3.txt	454	67 (10x10)	0.3114	36.0782	Time Out
input_4.txt	5242	526 (30x30)	2.9471	3855.8564	Time Out

Table 2: Performance Comparison of Solving Methods on Different Inputs

Based on the table, we can see that:

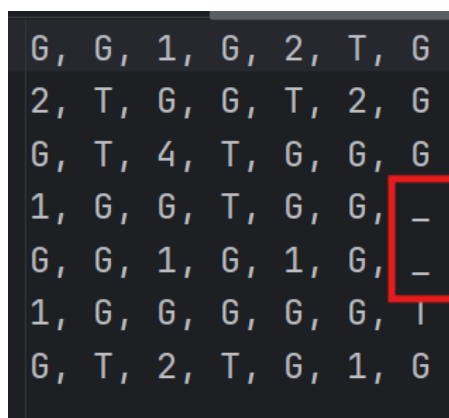
1. PySAT: consistently delivers the solution in the quickest times across all inputs, ranging from 0.0758ms to 2.9471ms, even on the largest input with grid that has size of 30x30 and

total of 5242 clauses and 526 empty cells. Scales efficiently with problem size, showcasing the strength of SAT solvers.

2. Backtracking: solves all inputs but with significantly higher runtime, especially for large instances:
 - Small inputs (e.g., `input_1.txt`) remain fast (`0.9174ms`),
 - But performance degrades steeply with size, reaching approximately `3855ms` on the largest input.
3. Brute-force: only solves the smallest input (`input_1.txt`) in a reasonable time (`2.0819ms`), and **times out** on all others, confirming its impracticality for anything beyond trivial cases.

4.2 Cells out of scope

Some inputs may lack sufficient information to determine the status of cells that are not within any 3×3 region surrounding a numbered cell. As a result, these cells cannot be conclusively identified as either a **TRAP** or a **GEM**, so they are marked as "_" to indicate unknown or out-of-scope cells.



G	G	1	G	2	T	G
2	T	G	G	T	2	G
G	T	4	T	G	G	G
1	G	G	T	G	G	_
G	G	1	G	1	G	_
1	G	G	G	G	G	1
G	T	2	T	G	1	G

Figure 1: those cells didn't adjacent any numbered cells thus the uncertainty

An alternative approach would be to assume that such cells are TRAPs by default, but I chose to preserve "_" to explicitly represent uncertainty.

4.3 Different results between solvers

Different solvers may yield different models, as the solution can depend on the specific input and solver behaviors. However, all resulting models are satisfiable and semantically correct,

with only minor differences in variable assignments. For example: the following functions show some minor differences between results from the PYSAT solver and the BACKTRACK solver.

PYSAT		BACKTRACK
G G 1 G 2 T G		T G 1 G 2 T T
2 T G G T 2 G		2 G G T G 2 G
G T 4 T G G G		G T 4 T G G G
1 G G T G G _		1 G T G T G _
G G 1 G 1 G _		G G 1 G 1 G _
1 G G G G G T		1 G G G G G T
G T 2 T G 1 G		G T 2 T G 1 G

Figure 2: Different results between PYSAT/BACKTRACK on input \mathcal{Q}_{at}

References

- [1] Wikipedia contributors. *DPLL algorithm*. Accessed: 2025-04-07. URL: https://en.wikipedia.org/wiki/DPLL_algorithm.
- [2] Nguyễn Hải Minh Nguyễn Ngọc Thảo. *2025 Lecture 05: PL-based Logical Agents*. HCMUS. DPLL algorithm Pages 81-83. 2025.
- [3] PySAT Team. *PySAT: SAT-based problem solving in Python*. <https://pysathq.github.io/docs/html/>. Accessed: April 07, 2025.