UNIVERSITY OF SCIENCE
VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

FACULTY OF INFORMATION TECHNOLOGY

COURSE: AI FOUNDATION

LAB 01 - GEM HUNTER

Instructor:
Nguyen Hai Dang
Nguyen Tran Duy Minh
Nguyen Ngoc Thao
Nguyen Thanh Tinh

Student:
Le Trung Kien
(23127075 - 23CLC08)

Ho Chi Minh City, April 7th, 2024

Contents

1 Introduction

This report explains how I broke down the Gem Hunter problem to solve it using Conjunctive Normal Form (CNF) via three different approaches: Brute force, Backtracking and Pysat library. After giving an overview about the problem and describing the project organization, I will explain how to generate the CNF.

To solve the CNF, I move to the approaches. For each of them, I carefully describe how it works and note some key points in implementation. Regarding the results, I provide the graph generated from the solving times of the three approaches. In addition, analysis is also included.

This project gives me a chance to have a practical view of how CNF works. By completing this lab, I gain a more profound understanding of CNF and solving logical problems.

2 Problem description

2.1 Rules

- The players explore a grid to find hidden gems while avoiding traps.
- Each tile with a number represents the number of traps surrounding it. (Number from 1 - 8).
- Each tile without a number represents the unknown position. Those positions might be Trap (T) or Gem (G).
- The task is to formulate the problem as CNF constraints and solve it using logic.

2.2 Goals

- Each nonnumerical tile in the grid must be assigned T for trap or G for gem.
- The output grid must satisfy all given numerical constraints, which means that that each number in each numbered tile must correctly reflect the number of adjacent traps.

3 Project organization

3.1 File

This section handles all the actions required with ".txt" files, including:

- Reading inputs from files.
- Writing outputs to files.
- Writing times to files.

3.2 Grid

This section manages the game's grid. It also has some essential functions:

- Valid position check to ensure that a position is in the grid.
- Getting neighbors' positions for further use
- Counting surrounding traps.
- Checking the correct position to ensure the number of surrounding traps is equal to the number of tile in that position.
- Checking if the grid is solved, which means the grid is satisfiable.

3.3 CNF handling

This section is responsible for handling with everything about CNF that my project needs. Here are the functions:

- Adding variables to the list of variables from blank tiles.
- Converting position to variable vice versa.
- Converting all the blank tiles to variables.
- Generating clauses and CNF (No duplicated clause).
- Getting all the variables from blank tiles of the CNF (no duplication).
- Checking if a clause or the whole CNF clauses are satisfiable.

3.4 Solving agents

There are 3 agents for 3 approach of solving the CNF clauses: brute force, backtracking (using dpll) and pysat.

4 CNF handling

4.1 CNF'S clauses and literals:

Before figuring out how to generate a CNF, let's look through some concepts. In PySAT, a literal should be represented as an integer: $x_n \rightarrow n$; $\neg x_n \rightarrow -n$. For instance, $x_1 \rightarrow 1$; $\neg x_1 \rightarrow -1$. In addition, a clause is a list of integers (literals). For example, $(\neg x_1, \neg x_2, x3) \rightarrow$ [-1, -2, 3]. CNF is a list of clauses, which means it is a matrix of integers. For instance, $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee \neg x_4) \rightarrow [[-1, 2], [1, 3, -4]]$ [1].

4.2 Correct logical principles for generating CNF

At first, I use dictionary to convert all the blank tiles into variables with increasing indices. To ensure that each tile with a number has exactly the number of surrounding traps, the tile must create a principle. A tile has a number of k surrounding traps means that it must obviously have exactly k surrounding traps. We can turn it into logic:

$$\texttt{Exactly K} \Leftrightarrow (\texttt{No more than K} \wedge \texttt{At least K}) \tag{4.1}$$

4.2.1 No more than K

Supposed we have a list of variables generated from surrounding neighbors (blank tiles) of the target tile called *neighbors_vars* with the size of n. Then, we have a logical sentence like this:

$$\texttt{No more than K} = \bigwedge[\neg a_1 \vee \neg a_2 \vee ... \vee \neg a_{k+1}], \tag{4.2}$$
$$\forall a_1, a_2, ...a_{k+1} \in neighbors\_vars, \forall i, j, \in [1, k+1], a_i \neq a_j$$

It works because if there are k + 1 traps (k + 1 variables are True), then there will be a clause with size of k + 1 turns into False, leading to the whole `No more than K` of logical sentence 4.1 become False. As a result, `Exactly K` is False.

4.2.2 At least K

Again, supposed we have a list of variables generated from surrounding neighbors (blank tiles) of the target tile called neighbors vars with the size of n. Then, we have a logical sentence like this:

$$\texttt{At least K} = \bigwedge[a_1 \vee a_2 \vee ... \vee a_{n-k+1}], \tag{4.3}$$
$$\forall a_1, a_2, ...a_{n-k+1} \in neighbors\_vars, \forall i, j, \in [1, k+1], a_i \neq a_j$$

It works because if there are k traps, there will be n - k gems. Each set of $a_1, a_2...a_{n-k+1}$ must include 1 trap and n - k gems. As all the gems are assigned with False, if there are not at least k traps, one clause will have only False literals, leading to False clause. Therefore, `At least K` and `Exactly K` of logical sentence 4.1 are False.

4.3 Generating clause

Using the logical principles in 4.2, we can easily generate a clause from list of blank neighbors variables and the number of surrounding traps of each tile.

4.3.1 Generating CNF automatically

Now, we can generate a CNF simply by visiting all the tiles of the grid and generating clause for each tile with number. Then add those clauses into the CNF.

5 Solving approaches

All the approaches recieve a CNF as an input and then return a model which is a list of integers. The integers in the model represent each value is True (Trap) or False (Gem) following the concepts included in section 2.

5.1 Pysat library approach

- First we create a solver, which is the Solver() provied by Pysat library
- The solver gets the CNF.
- The solver solves the CNF to get the model.
- The function returns the model if there is solved sucessfully. Else, the it returns None.

5.2 Brute force algorithm approach

- Getting the length n of the variables (empty tiles), so there will be 2^n cases.
- Assign 2^n into a variable called *total*.
- We recognize that the bits representation of each number in [0, total - 1] can represents a model. Therefore, we can get all the cases by visiting all the numbers in range [0, total - 1].
- For each model, we check if it is satisfiable or not. If it is satisfiable, return the model.
- For huge test cases, the solving time of Brute force is absolutely "crazy", so I make a number of attempts to limit the models exploration. This number is 2 million at the beginning and reduces by 1 for each time creating a model. If that number goes down to 0, the function returns None and stop solving.

5.3 Backtracking algorithm approach

I refer to the DPLL implementation of David Fernig [2]. Here is how I implement backtracking algorithm using DPLL algorithm.

- Return True and the assignments if there is nothing left in the CNF, which means the CNF is now satisfiable.
- Return False if there is any clause of the CNF is empty.

- Get the list of unassigned variables (not in the assignments). If the list is empty, there is no solution because the CNF is still there but there is no more variable to assign.
- Get the first variable of the unassigned list to assign value True or False and called it l because we will check the literal related to that variable later.
- For each assignment of True and False of the first variable, we do the following steps
  - Declare a new empty CNF.
  - For every clause in the old CNF, if it has literal l and satisfy the assignment, we won't add this clause to the new CNF because it is True. We can also say that this clause is simplified.
  - We create a new clause from the old clause by eliminate all the literal that do not satisfy the assignment of literal l. For insatnce, if l is x_1, then $(\neg x_1 \vee x_2 \vee x_3) \wedge x_1 \Leftrightarrow (x_1 \wedge (x_2 \vee x_3)$. So $(x_2 \vee x_3)$ is the new clause. Add the new clause into the new CNF. If the new clause if empty, it leads to the return in step 2. The reason is that the old clause only has the literal that does not satisfy the assignment of l. For instance, l is x_1, the old clause is $\neg x_1$, then $x_1 \wedge \neg x_1 \Leftrightarrow \{\}$.
  - Then we call recursive backtracking with the new cnf and the new assignments and assign the return value into result.
  - If the result is satisfiable (True), the final assignment is the result.
  - Return True with that final assignments.
- If after the first variable assignment step, the result is still not satisfiable, return False.
- To avoid huge test cases, I also give backtracking a number of attempts to limit the exploration like Brute force. This number reduces for every recursive call.

6 Experiments

6.1 Input and Output Format

- All the input files are located in the input directory. The name of each of them follows the format "input_x.txt" with x is the number of input.
- All the output files are located in the output directory. The name of each of them follows the format "output_x.txt" with x is the number of output. Each output file contains the solved grid of 3 approaches: Pysat library, Brute force and backtracking.
- File times.txt contains all the solving times of the 3 mentioned approaches for all test cases.

Input file

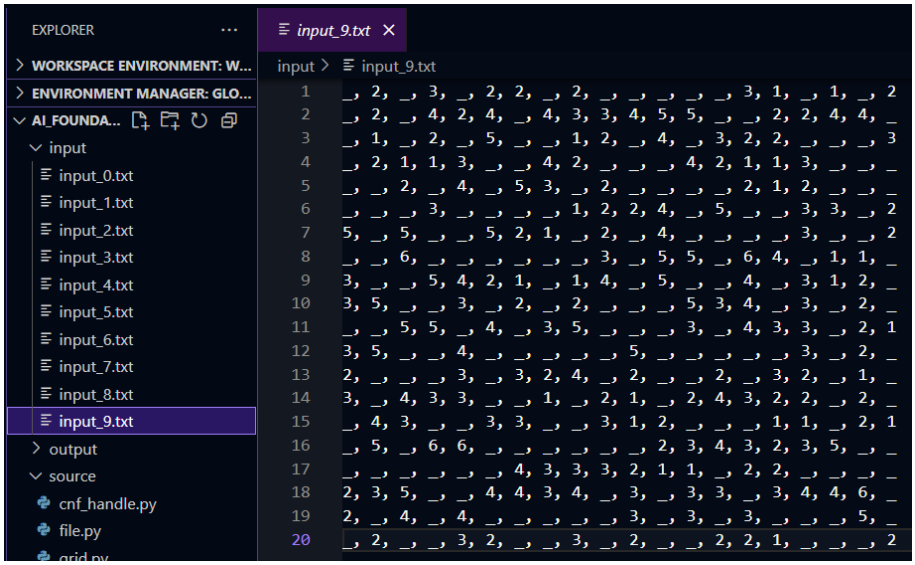

Figure 1: Image of input file for test case 9

Output file

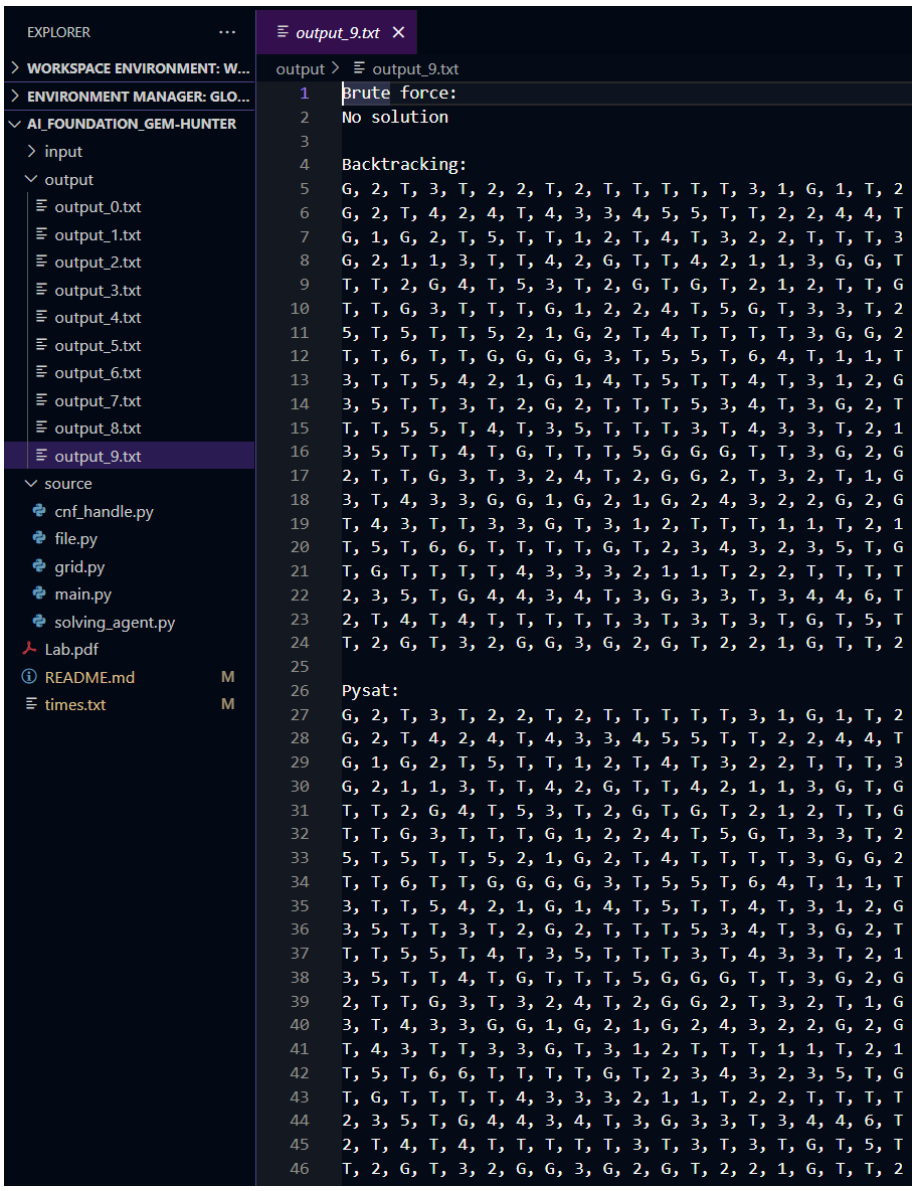

Figure 2: Image of output file for test case 9

6.2 Result

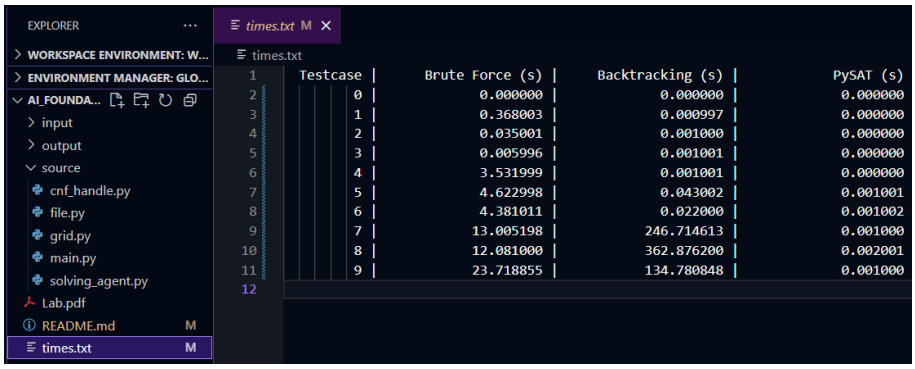

Figure 3: Image of solving times output file

Please note that brute force and backtracking failed from test case 4 and 7, respectively as they reached the limited number of attempts. The solving time in the time output file of them are just the total time to reach the limited number of attempts that is mentioned in 2 previous sections 5.2 and 5.3.

6.3 Analysis

From the image of solving times 3 of the 3 approaches, we remove the unsolved result and receive the table below.

Testcase	Brute Force (s)	Backtracking (s)	PySAT (s)
0	0.000000	0.000000	0.000000
1	0.368003	0.000997	0.000000
2	0.035001	0.001000	0.000000
3	0.005996	0.001001	0.000000
4	unsolved	0.001001	0.000000
5	unsolved	0.043002	0.001001
6	unsolved	0.022000	0.000000
7	unsolved	unsolved	0.001000
8	unsolved	unsolved	0.002001
9	unsolved	unsolved	0.001000

Table 1: Performance comparison of Brute Force, Backtracking, and PySAT algorithms

We get the graph below from the table above.

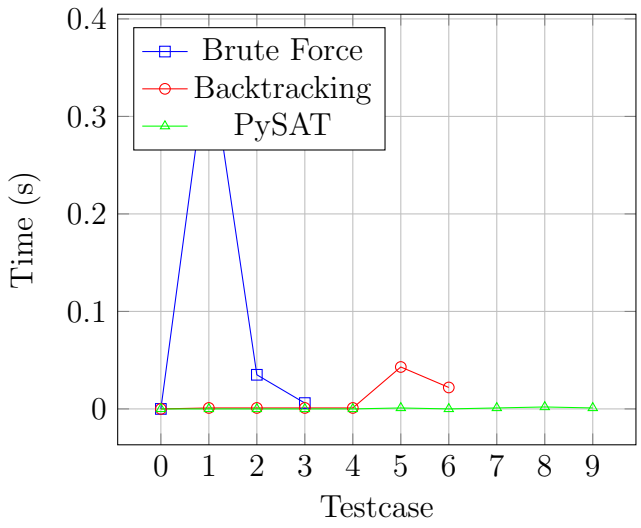

Figure 4: Performance comparison of Brute Force, Backtracking, and PySAT algorithms

It is obvious that Brute force is a poor approach that failed from the 4^{th} test case

with the size of 11×11. In addition, Backtracking is better because it can solve until the 6^{th} and test case 9^{th}. This proved backtracking can still solve 20×20 if it can simplify the CNF effectively.

Brute force is a greedy algorithm that lead to long solving time. Indisputably, it always takes the longest time to solve the CNF. The time complexity of Brute force algorithm for this problem is $O(2^n)$.

Backtracking can simplify the CNF so it does not need to explore the whole models. This make it the second fastest approach of all three. Finally, the solver provided by the Pysat library is the most optimal one with the rapid solving speed.

7 Source code and Video demo

7.1 Link

- Github repository: $https://github.com/Hacdess/AI_FoundationGem-hunter.git$
- Youtube video: $https://www.youtube.com/watch?v=m0_XZO_Wrbc$

ubsectionHow to Run Source Code

- Prepare your input files in the `input` directory. The input files should be named `input_x.txt` where x is the number of the input.
- Run the main program:
  1. **Method 1**: Navigate to the `AI_Foundation_Gem-hunter` directory and use the command: `python source/main.py`
  2. **Method 2**: Open the `main.py` file and use the "Run" button in your IDE.
- The output will be saved in the `output` directory as `output_x.txt` corresponding to each input file.

8 Progress

No.	Criteria	Done
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%

Table 2: Task Completion Checklist

References

[1] Antonio Morgado Alexey Ignatiev Joao Marques-Silva. *Boolean formula manipulation (pysat.formula)*. Accessed: March 2025. URL: https://pysathq.github.io/docs/html/api/formula.html?highlight=append#module-pysat.formula.

[2] David Fernig. *dpll*. Accessed: March 2025. URL: https://gist.github.com/davefernig/e670bda722d558817f2ba0e90ebce66f.