University of Science - VNUHCM

Faculty of Information Technology

PROJECT REPORT

Topic: Gem hunter

Course: Artificial Intelligence

Student:

Ho Minh Dang - 22127050

Vo Hung Khoa - 22127202

Nguyen Binh Minh - 2217266

Vo Huu Tuan - 22127439

Lecturer:

Le Ngoc Thanh

Nguyen Ngoc Thao

Nguyen Hai Dang

Nguyen Tran Duy Minh

Contents

1 Information

1.1 Member Information

No.	ID	Name	Note
1	22127050	Ho Minh Dang	
2	22127202	Vo Hung Khoa	**Leader**
3	22127266	Nguyen Binh Minh	
4	22127439	Vo Huu Tuan	

Table 1: Student Information

1.2 Project Information

Name project	Gem Hunter
Environment	Python 3.10.11
Graphic library	Tkinter + PIL
IDE	VScode

Table 2: Project information table

1.3 Checklist

No.	Specifications Scores	Member	Progress (%)
1	Logical principles for generating CNFs.	Dang & Minh	100%
2	Generate CNFs automatically	Dang	100%
3	Solve CNF using Pysat library	Dang	100%
4	Implement an optimal algorithm without a library	Khoa	100%
5	Brute-force algorithm	Tuan	100%
6	Backtracking algorithm	Minh	100%
7	Generate map	Tuan	100%
8	Report	All members	100%

Table 3: Checklist table

2 Running Instruction

Prerequisites:

- Python installed.
- Pysat library installed.

This is a guide on how to run the project ”Gem Hunter” from source code.

1. First locate the ”GUI.py” file in the ”Source/Source_code” folder
2. Then open the file in Visual Studio code. On the upper right corner of the Visual Studio Code editor choose the down arrow symbol next to the Play symbol

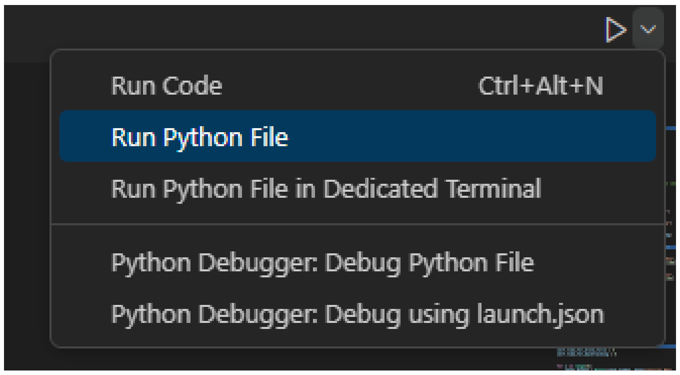

Run Code Ctrl+Alt+N
Run Python File
Run Python File in Dedicated Terminal
Python Debugger: Debug Python File
Python Debugger: Debug using launch.json

Figure 1: Button

3. Choose ”Run Python File” to execute the program.
4. (Alternative) Open the ”Source_code” folder in the terminal and run the command ”py GUI.py” to execute the program.
5. Proceed to watch the demo video in order to use the program.

3 Logical principles for generating CNFs

3.1 Example case

In order to create a CNF formula, we need to take a closer look in to a case.

Example case:

X1 = Trap	X2 = Trap	X3
X8	Number = 2	X4
X7	X6	X5

Table 4: 2 traps

Let X1, X2 are traps, the other Xi is anything but trap. To make this case true the CNF must be:

$$(X1 \land X2) \Leftrightarrow (\neg X3 \land \neg X4 \land \ldots \land \neg X8)$$

This CNF means that if X1 and X2 are "Trap" then the others are not and vice versa.

But as you can see, turning this logic into code is not easy. Therefore we need to reformulate the logic above into something much easier to code:

$$\begin{aligned}
&\equiv [(X1 \land X2) \Rightarrow (\neg X3 \land \neg X4 \land \ldots \land \neg X8)] \land [(\neg X3 \land \neg X4 \land \ldots \land \neg X8) \Rightarrow (X1 \land X2)] \\
&\equiv [\neg(X1 \land X2) \lor (\neg X3 \land \neg X4 \land \ldots \land \neg X8)] \land [(X3 \lor X4 \lor \ldots \lor X8) \lor (X1 \land X2)] \\
&\equiv [(\neg X1 \lor \neg X2) \lor (\neg X3 \land \neg X4 \land \ldots \land \neg X8)] \land [(X3 \lor X4 \lor \ldots \lor X8) \lor (X1 \land X2)] \\
&\equiv [(\neg X1 \lor \neg X2 \lor \neg X3) \land (\neg X1 \lor \neg X2 \lor \neg X4) \land \ldots \land (\neg X1 \lor \neg X2 \lor \neg X8)] \\
&\qquad \land [(X1 \lor X3 \lor X4 \lor X5 \lor \ldots \lor X8) \land (X2 \lor X3 \lor X4 \lor X5 \lor \ldots \lor X8)]
\end{aligned}$$

Let explain each part:

- $(\neg X1 \lor \neg X2 \lor \neg Xi)$: with i from 3 to 8, this clause only false if there are more than 3 traps.
- $(Xy \lor X3 \lor X4 \lor X5 \lor \ldots \lor X8)$: with y from 1 to 2, the clause only false if there are no trap at all

We have successfully turn the original CNF into multiple small clauses.

In the above example case, there are 2 traps at the precise location 1, 2. What happens if we know only the amount of traps but not the location?

3.2 2-trap CNF

To generate CNF for unknown number of trap location, we call the first part is L and the second part is U:

$$CNF = L \wedge U.$$

- L = $(\neg Xa \vee \neg Xb \vee \neg Xc)$; the number of possible combinations of 3 variables Xa, Xb, Xc from a set of 8 variables is $\binom{8}{3}$
- U = $(Xa \vee Xb \vee Xc \vee Xd \vee Xe \vee Xf \vee Xg)$; the number of possible combinations of 7 variables Xa, Xb, Xc, Xd, Xe, Xf, Xg from a set of 8 variables is $\binom{8}{7}$

We have successfully generalized a 2-trap CNF. But how to make 3-trap CNF, 4-trap CNF?

3.3 k-trap CNF

Formulate the general k-trap CNF logic:

$$CNF(k, n) = L(k + 1) \wedge U(n - k + 1)$$

Let CNF(k, n) with k is the number of traps in n surrounding cells.

- $L(k+1)$: $(k+1)$-combination of n cells taken from $\{Xa, \ldots, Xn\}$ to create $(\neg Xx \vee \ldots \vee \neg Xy)$
- $U(n - k + 1)$: $(n - k + 1)$-combination of n cells taken from $\{Xa, \ldots, Xn\}$ to create $(Xx \vee \ldots \vee Xy)$

4 Brute-force algorithm

4.1 Initial idea

Initially, the brute-force algorithm is based on the state of each unknown-cell (cells marked with "_"): Each unknown-cell can either be a trap or a gem, meaning each unknown-cell has 2 states. Therefore, we need to check all possible states of the $N * N$ cells in the map to find the solution! According to this calculation, in the worst-case, we have to consider all 2^{N^2} states. This is a huge problem because, for N = 10, the total number of states is $2^{100} \approx 1.26 * 10^{30}$, an enormous number! It is a very large number, not to mention the case of N = 20, it is $2^{400} \approx 2.5 * 10^{120}$ states. Surely, this poses a significant challenge for a computer!

4.2 Better ideas to implement

Therefore, the new idea for the brute-force algorithm is to consider the number of unexplored cells based on the number-cells (cells with integer values). The idea is as follows:

- The unknown-cell with the highest number of number-cells around will be marked as a trap.
- Decrease the value of the number-cells by 1 if there is a neighboring cell marked as a trap.
- If there is a number cell with a value of 0, the unexplored cell near it will be marked as a gem.

With the simple idea above, the algorithm will work as follows:

1. Iterate through all cells in the map:
   (a) If the cell is unknown: Perform step 2
   (b) Otherwise, skip
2. Check the neighboring cells of the current cell:
   (a) If there is a number-cell with a value of 0: Mark the current cell as a gem and move to the next cell
   (b) Otherwise, increase the count of number-cells of the current cell by 1

3. After iterating through all cells, the cell with the highest number of neighboring number-cells will be marked as a trap, and the value of all neighboring number-cells will be decreased by 1.

4. Repeat the process until there are no unknown-cells left.

With the above operating method, the complexity of the algorithm is significantly reduced! We only need to traverse the map based on the number of unexplored cells (less than N * N cells), each cell needs to check 8 neighboring cells and decrease the value of the neighboring number-cells after each iteration. Therefore, the worst-case will have a complexity of $O(N) = N * N * (N * N * 8 + 8)$. With N = 20, $O(20) = 1,280,032,000 = 1,28 * 10^{10}$, faster than the initial idea by nearly 10^{110} times!

4.3 Pros and Cons

Pros:

- Simple idea: easy to understand and implement.
- Completeness: It is guaranteed to find a satisfying assignment if one exists.

Cons:

- Worst-case Time Complexity: The algorithm can become prohibitively slow for maps with large sizes.
- Memory Efficiency: The algorithm consumes a lot of memory space for memorization when iterating over unknown-cells

5 Backtracking algorithm: DPLL

Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.

5.1 Function

5.1.1 unit propagate function

The purpose of the unit_propagate function is to simplify the given CNF (Conjunctive Normal Form) formula by identifying and propagating unit clauses.
A unit clause is a clause that contains only one unassigned literal.

5.1.2 DPLL

This is an step by step illustration of how DPLL - iterative version works.

1. Initialization: Initialize a stack with the initial CNF formula and an empty set of variable assignments. Start with an empty stack.
2. Main Loop: While the stack is not empty, repeat the following steps:
3. Unit Propagation: Pop a formula and assignments from the top of the stack. Perform unit propagation on the formula with the current assignments.
4. Conflict Check: If there are any empty clauses (indicating a conflict), skip to the next iteration of the loop.
5. Satisfiability Check: If the formula becomes empty after unit propagation (all clauses are satisfied), return True along with the satisfying assignment.
6. Variable Selection: If there are unassigned literals remaining, choose a literal (non-deterministically in this implementation).
7. Branching: Create two new branches by assigning the chosen literal to True and False, respectively.

8. Push Branches onto Stack: Push these two new branches (modified formulas and assignments) onto the stack.
9. Termination: If the loop terminates without finding a satisfying assignment, return False.
10. Backtracking: Backtracking is implicitly handled by the stack. If a branch fails (returns False), the algorithm simply continues with the next branch on the stack.

5.2 Pros and Cons

Pros:

- Completeness: It is guaranteed to find a satisfying assignment if one exists.
- Efficiency in Practice: While the worst-case time complexity of DPLL is exponential. Various optimizations, such as unit propagation, pure literal elimination, and efficient branching strategies, can significantly improve performance.
- Memory Efficiency: The iterative version of DPLL can be memory-efficient, as it avoids the overhead of recursive function calls by using a stack to maintain state.

Cons:

- Exponential Worst-case Time Complexity: The algorithm can become prohibitively slow for very large or complex CNF formulas.
- Limited Performance Guarantees: While DPLL is complete, it does not provide any performance guarantees in terms of runtime.
- Difficulty with Hard Instances: There are certain classes of SAT instances known as "hard" instances for which DPLL may struggle to find a solution within a reasonable amount of time.
- Example for a hard instance:
  - In level 4, the map 20x20 has an interesting layout that makes the DPLL algorithm runs extremely slower that other algorithm, even the brute force algorithm.

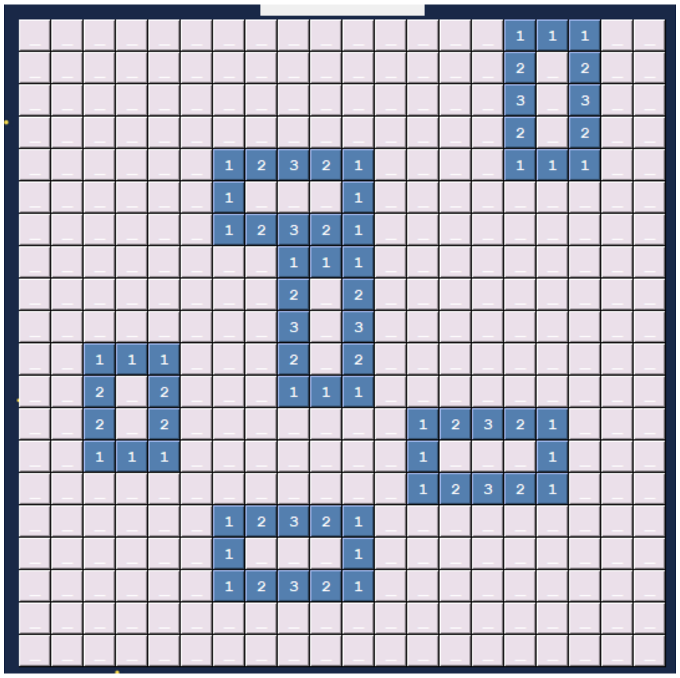
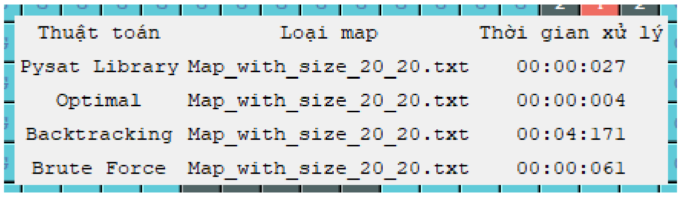


Figure 2: 20x20 map

– The DPLL algorithm performed much worse than any other algorithm because the constantly back tracking to find the satisfied condition.



Thuật toán	Loại map	Thời gian xử lý
Pysat Library	Map_with_size_20_20.txt	00:00:027
Optimal	Map_with_size_20_20.txt	00:00:004
Backtracking	Map_with_size_20_20.txt	00:04:171
Brute Force	Map_with_size_20_20.txt	00:00:061

Figure 3: DPLL runtime compare to other

6 Optimal Algorithm

To solve this problem, we have chosen a basic backtracking technique to assign variables and eliminate invalid assignments. To make the variable assignment more efficient, we additionally use the Jeroslow-Wang (JW) heuristic function to achieve that.

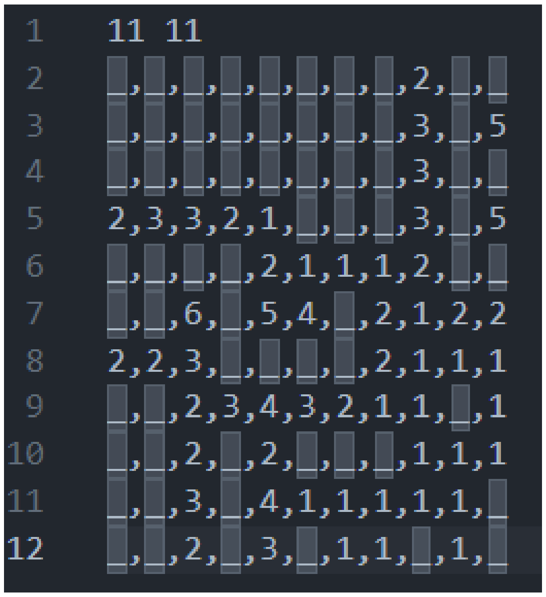
6.1 Introduction

The Jeroslow-Wang (JW) heuristic is used to prioritize variable assignment during the solving process of the Boolean Satisfiability Problem (SAT). It assesses the importance of each variable based on its frequency of occurrence in positive literals across the Conjunctive Normal Form (CNF) clauses. The JW score of a variable is calculated by summing the exponential of the negative length of each clause containing the variable. Variables with higher JW scores are given priority during variable selection, as they are more likely to contribute significantly to finding a valid assignment, thus enhancing the efficiency of the SAT-solving algorithm.

6.2 Function overview

6.2.1 Input Grid Preparation

- The algorithm takes the input grid representing the Minesweeper game and its size.



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

Figure 4: Structure of an input file

6.2.2 CNF Clause Generation

- For each cell in the grid, the algorithm generates CNF (Conjunctive Normal Form) clauses based on the Minesweeper rules.
- It identifies neighboring cells and generates clauses to satisfy the numeric clues provided in the grid.

6.2.3 SAT Solving with Heuristic

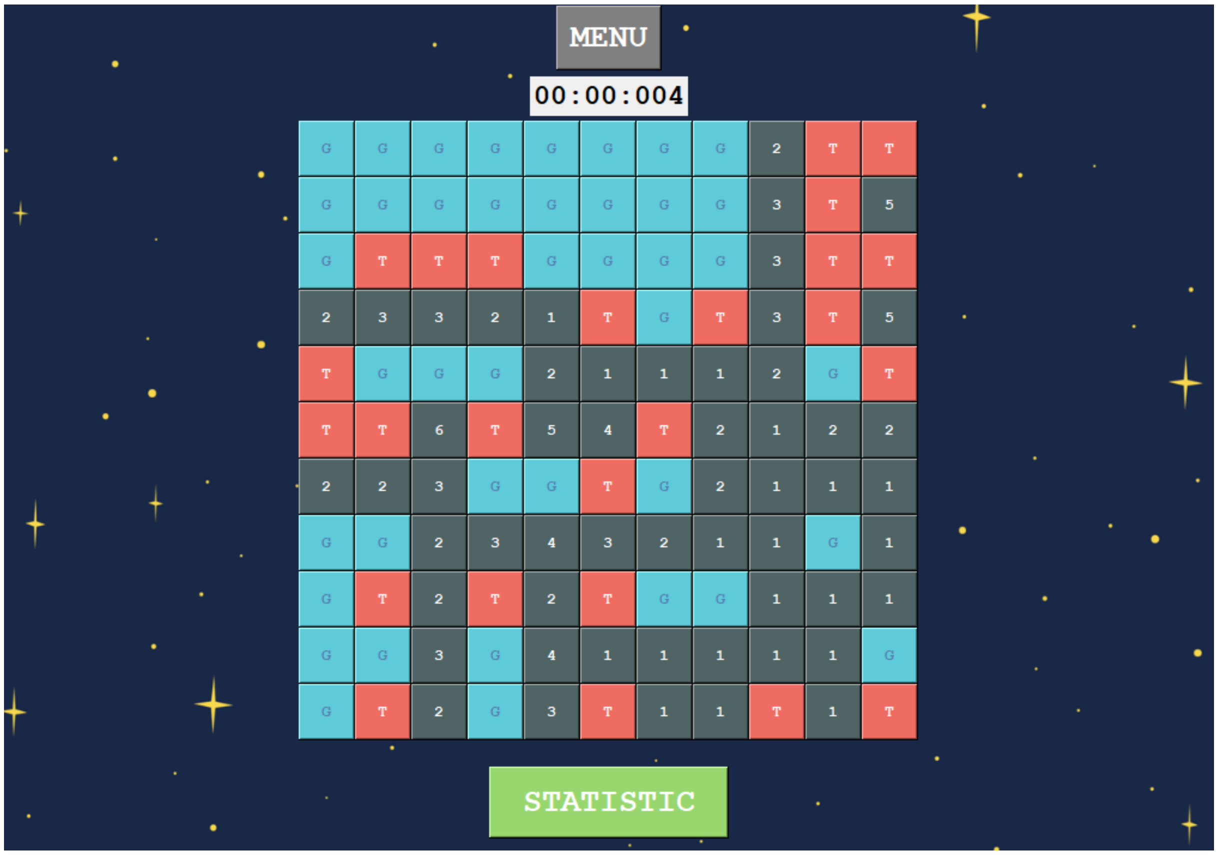
- The algorithm attempts to solve the SAT (Boolean Satisfiability Problem) generated from the CNF clauses.
- It employs backtracking along with the Jeroslow-Wang (JW) heuristic to efficiently explore the solution space.
- Backtracking involves recursively assigning truth values to variables and backtracking when a contradiction is encountered.
- The JW heuristic guides variable selection, prioritizing variables with higher Jeroslow-Wang scores.

6.2.4 Solution Processing

- If a satisfying assignment is found, the algorithm processes the assignment to generate the solution grid.
- It maps the assignment to the cells of the original grid, marking mines ('T') or safe cells ('G') accordingly.
- If no satisfying assignment is found, the algorithm concludes that no solution exists.

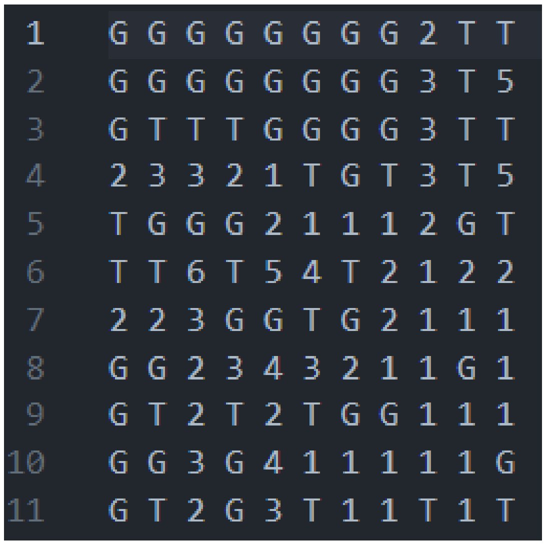
6.2.5 Ouput

- The algorithm returns the solution grid if a solution is found.
- If no solution is found, it indicates the absence of a valid solution.



```
G G G G G G G G 2 T T
G G G G G G G G 3 T 5
G T T T G G G G 3 T T
2 3 3 2 1 T G T 3 T 5
T G G G 2 1 1 1 2 G T
T T 6 T 5 4 T 2 1 2 2
2 2 3 G G T G 2 1 1 1
G G 2 3 4 3 2 1 1 G 1
G T 2 T 2 T G G 1 1 1
G G 3 G 4 1 1 1 1 1 G
G T 2 G 3 T 1 1 T 1 T
```

Figure 5: Display result



```
G G G G G G G G 2 T T
G G G G G G G G 3 T 5
G T T T G G G G 3 T T
2 3 3 2 1 T G T 3 T 5
T G G G 2 1 1 1 2 G T
T T 6 T 5 4 T 2 1 2 2
2 2 3 G G T G 2 1 1 1
G G 2 3 4 3 2 1 1 G 1
G T 2 T 2 T G G 1 1 1
G G 3 G 4 1 1 1 1 1 G
G T 2 G 3 T 1 1 T 1 T
```

Figure 6: The output is also saved in a file in folder "output"

6.3 Pros and Cons

6.3.1 Pros

- Easy Implementation: The algorithm utilizes basic backtracking and simple heuristics like JW, making it easy to implement and understand.
- Problem-solving Capability: The algorithm can solve the Minesweeper problem by finding a solution that satisfies the Boolean constraints described.
- Flexibility: Other heuristic functions can be modified or added to improve the algorithm's performance.

6.3.2 Cons

- Suboptimal Performance: Using basic backtracking and JW heuristics may result in suboptimal performance, especially when solving large or complex problems.
- Incomplete Search Capability: Due to the use of backtracking, the algorithm may fail to find a solution within a reasonable time for large-sized problems.
- Potential for Infinite Loops: In some cases, the algorithm may run into infinite loops if it cannot find a solution or eliminate invalid situations.

6.4 Conclusion

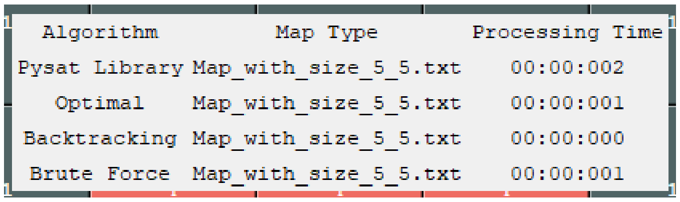
- The algorithm sequentially performs grid preparation, CNF clause generation, SAT solving with backtracking and heuristic, solution processing, and output generation.
- By combining backtracking with the JW heuristic, the algorithm efficiently solves the Minesweeper game, providing accurate solutions when possible.
- Generally, due to its use of basic backtracking, this algorithm may be less optimized compared to commonly used algorithms like DPLL or libraries like Pysat. Therefore, its processing time might vary, especially in some scenarios.

No.	Map name	Pysat	Optimal	Backtracking	Brute_force
1	Map_with_size_5_5	00:00:002	00:00:001	00:00:000	00:00:001
2	Map_with_size_9_9	00:00:002	00:00:006	00:00:001	00:00:004
3	Map_with_size_11_11	00:00:003	00:00:011	00:00:004	00:00:011
4	Map_with_size_20_20	00:00:005	00:00:059	00:02:796	00:00:030
5	Map_with_size_25_25	00:00:012	00:00:210	00:00:060	00:00:240

Table 5: The processing time statistics of the 4 algorithms

7 Running Time Comparison

7.1 Level 1



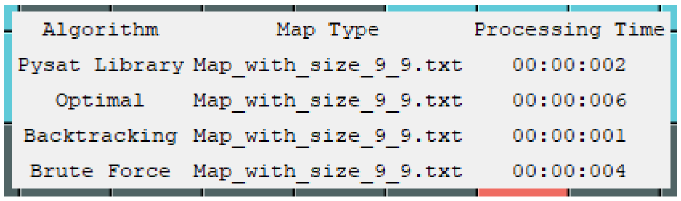
Algorithm	Map Type	Processing Time
Pysat Library	Map_with_size_5_5.txt	00:00:002
Optimal	Map_with_size_5_5.txt	00:00:001
Backtracking	Map_with_size_5_5.txt	00:00:000
Brute Force	Map_with_size_5_5.txt	00:00:001

Figure 7: Level 1 Running Time

Observation: Because of the small size of this level(5x5), every algorithm running time is extremely low.

7.2 Level 2

The size of the map is 9x9.



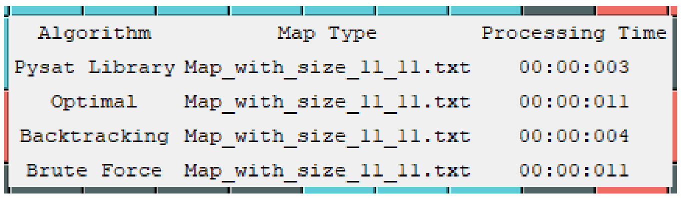
Algorithm	Map Type	Processing Time
Pysat Library	Map_with_size_9_9.txt	00:00:002
Optimal	Map_with_size_9_9.txt	00:00:006
Backtracking	Map_with_size_9_9.txt	00:00:001
Brute Force	Map_with_size_9_9.txt	00:00:004

Figure 8: Level 2 Running Time

Observation: Even though it is called the "Optimal Algorithm", this algorithm is not fully developed to the finest of its level. Other algorithms work just fine.

7.3 Level 3

The map size is 11X11.



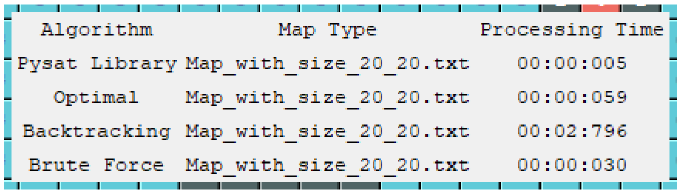
Algorithm	Map Type	Processing Time
Pysat Library	Map_with_size_11_11.txt	00:00:003
Optimal	Map_with_size_11_11.txt	00:00:011
Backtracking	Map_with_size_11_11.txt	00:00:004
Brute Force	Map_with_size_11_11.txt	00:00:011

Figure 9: Level 3 Running Time

Observation: As the map grows in size, the time it takes to solve the puzzle increases rapidly. In this level 3 map, the more information given (numbered cell) the quicker the backtracking algorithm runs.

7.4 Level 4

The map size is 20x20.



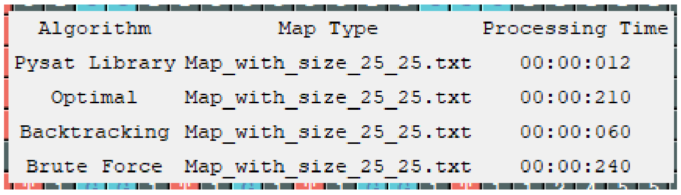
Algorithm	Map Type	Processing Time
Pysat Library	Map_with_size_20_20.txt	00:00:005
Optimal	Map_with_size_20_20.txt	00:00:059
Backtracking	Map_with_size_20_20.txt	00:02:796
Brute Force	Map_with_size_20_20.txt	00:00:030

Figure 10: Level 4 Running Time

Observation: As you can see, the map size is 20x20, which is really big. But the amount of numbered cells is very little. Because of the little information, the backtracking algorithm takes a ton of time to run. Therefore the higher the *numbered_cell/unknown_cell* ratio, the faster the DPLL algorithm.

7.5 Level 5

The map size is 25x25.



Algorithm	Map Type	Processing Time
Pysat Library	Map_with_size_25_25.txt	00:00:012
Optimal	Map_with_size_25_25.txt	00:00:210
Backtracking	Map_with_size_25_25.txt	00:00:060
Brute Force	Map_with_size_25_25.txt	00:00:240

Figure 11: Level 5 Running Time

Observation: In the end, the Pysat library is the fastest algorithm. The "Optimal Algorithm" our group proposes is not really good enough but it has room to improve. The backtracking algorithm works well as there are ton of information. Brute force is indeed the slowest one.

8 Additonal Information

You can review the source code and online video by clicking on the link below.

- Video
- Source

Reference

1. tkinter — Python interface to Tcl/Tk

2. DPLL algorithm

3. Phase Selection Heuristics for Satisfiability Solvers - Jingchao Chen