HO CHI MINH NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

---o0o---

Introduction to Artificial Intelligent

Project 02

Gem Hunter

Instructors:	Nguyễn Ngọc Thảo	Hồ Thị Thanh Tuyến
	Lê Ngọc Thành	Nguyễn Trần Duy Minh

Members:	Trần Anh Minh	22127275
	Đoàn Đặng Phương Nam	22127280
	Bùi Nguyễn Lan Vy	22127465
	Diệp Gia Huy	22127475

HO CHI MINH CITY, APRIL, 2024

CONTENTS

I. Assignment Planner

a. Team information

Full name	MSSV
Trần Anh Minh	22127275
Đoàn Đặng Phương Nam	22127280
Bùi Nguyễn Lan Vy	22127465
Diệp Gia Huy	22127475

b. Task division

Tasks	Members in charge
Generating CNF	All 4 members
Generating CNF automatically	Tran Anh Minh
Use PySAT library to solve CNF	Bui Nguyen Lan Vy
Implementing DPLL	Tran Anh Minh
Implementing GA	Doan Dang Phuong Nam
Implementing Brute Force	Diep Gia Huy
Implementing Back-tracking	Doan Dang Phuong Nam
Generating test cases	Bui Nguyen Lan Vy Diep Gia Huy
Performing comparisons	Tran Anh Minh Doan Dang Phuong Nam
Reporting	Tran Anh Minh Doan Dang Phuong Nam Bui Nguyen Lan Vy
Video demonstrating	Diep Gia Huy

c. Completion

Criteria	Completion
The logic for generating CNFs	100%
Generate CNFs automatically	100%
Use PySAT library to solve CNFs	100%
Implement an optimal solution – DPLL, GA	100%
Implement brute force, backtracking	100%
Generate test cases (Maps)	100%
Reporting	100%
Video demonstration	100%

II. Environment Requirement

Python version: 3.10+.

Modules/Library: pySAT.
Consider installing 'pySAT' using *pip install python-sat* if it is not available.

Usage: Change the working directory to the SOURCE folder.
Then run the application by executing *python main.py* in the console/terminal.
Notice: The command may vary across platforms, the above command it tested on a Windows operating system.

We highly recommend running the program within a big terminal window because the maps printed to the screen are displayed in a custom-designed table and may take up some space, which is hard to follow along with the process.

III. Introduction

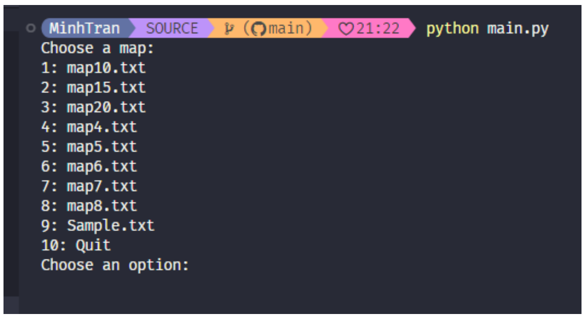
In this report, we present a few methods for solving the mentioned Gem Hunter strategic problem.

After doing research, we realized that many optimal algorithms to solve SAT problems used backtracking as their basis. Therefore, we chose to implement **DPLL** as both an **optimal algorithm** and a **backtracking solution**. Moreover, we also implement **GA**, as it is **another optimal solution** to solve SAT problems **without backtracking**.

Instruction video: *Demonstration Video*

First, locate the *main.py* file in the *SOURCE* folder and run it by executing python main.py in the terminal.

A list of maps that have already been prepared inside the *Testcase* folder will be displayed.

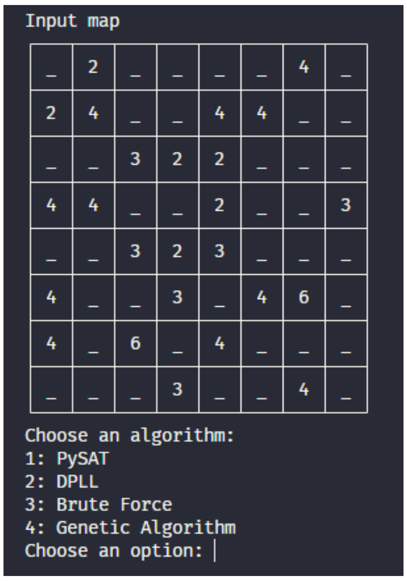


```
MinhTran  SOURCE  (main)  21:22  python main.py
Choose a map:
1: map10.txt
2: map15.txt
3: map20.txt
4: map4.txt
5: map5.txt
6: map6.txt
7: map7.txt
8: map8.txt
9: Sample.txt
10: Quit
Choose an option:
```

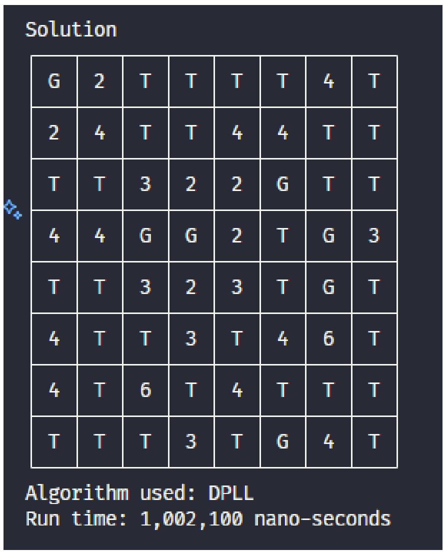
The chosen map will be printed on the screen in box format.

After that, you can choose your preferred method to solve the selected puzzle. There are 4 options: using the PySAT library, DPLL, GA, or Brute Force.

When the algorithm is done running, the running time and the solution will show up, and a solution file is also created inside the Solution folder.



```
Input map
| _ | 2 | _ | _ | _ | _ | 4 | _ |
| 2 | 4 | _ | _ | 4 | 4 | _ | _ |
| _ | _ | 3 | 2 | 2 | _ | _ | _ |
| 4 | 4 | _ | _ | 2 | _ | _ | 3 |
| _ | _ | 3 | 2 | 3 | _ | _ | _ |
| 4 | _ | _ | 3 | _ | 4 | 6 | _ |
| 4 | _ | 6 | _ | 4 | _ | _ | _ |
| _ | _ | _ | 3 | _ | _ | 4 | _ |
Choose an algorithm:
1: PySAT
2: DPLL
3: Brute Force
4: Genetic Algorithm
Choose an option:
```



```
Solution
| G | 2 | T | T | T | T | 4 | T |
| 2 | 4 | T | T | 4 | 4 | T | T |
| T | T | 3 | 2 | 2 | G | T | T |
| 4 | 4 | G | G | 2 | T | G | 3 |
| T | T | 3 | 2 | 3 | T | G | T |
| 4 | T | T | 3 | T | 4 | 6 | T |
| 4 | T | 6 | T | 4 | T | T | T |
| T | T | T | 3 | T | G | 4 | T |
Algorithm used: DPLL
Run time: 1,002,100 nano-seconds
```

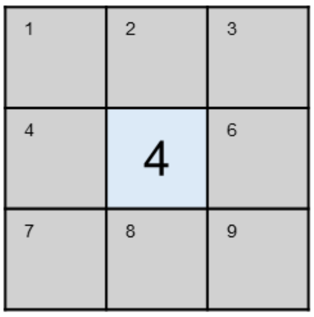
IV. Algorithms

1. *Logical principles in generating CNF*

If a certain location on the map is a trap, we will set that cell to be true; Otherwise, it will be false.

In the game, there are some conditions we need to examine:

- Every cell is either a trap or not a trap.
- The numbered cells are neither traps nor gems.
- Observing the numbered cell, the total number of traps in the surrounding unlabeled cells must be equal to that cell's value.

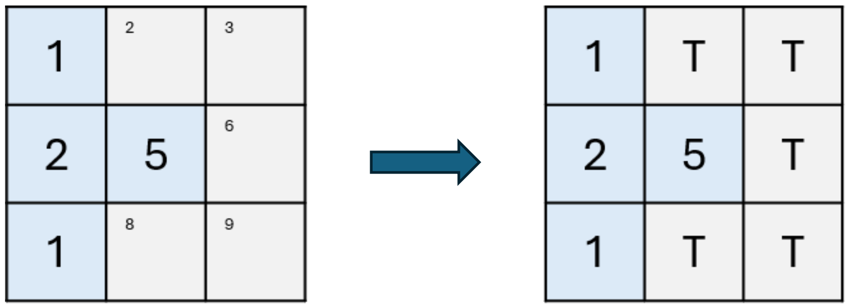


E.g.: There is traps in 4 of the remaining cells (1, 2, 3, 4, 6, 7, 8, 9)
Assuming the trap cells are 1, 2, 3, 4, the CNF clause we achieve is 1 ∨ 2 ∨ 3 ∨ 4.

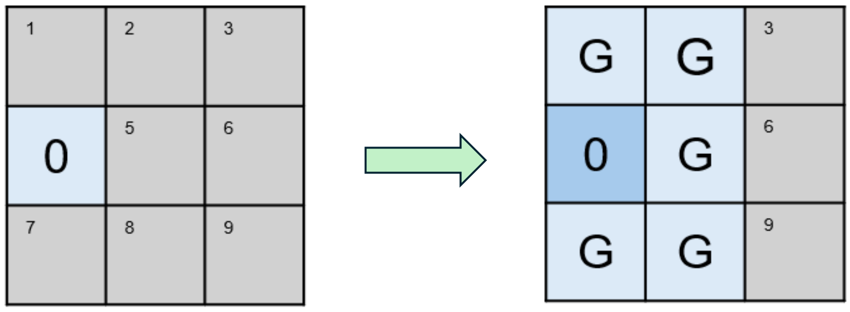
Therefore, we can infer two more principles (supposing the cell's value is n and the number of surrounding unassigned cells is u):

- The area must have at most n traps. In other words, considering every combination of $n + 1$ cells in the area, it is guaranteed to have at least one safe cell (gem).
  - *CNF clauses:* 1 ∨ 2 ∨ 3 ∨ 4 ∨ 6, 1 ∨ 2 ∨ 4 ∨ 6 ∨ 7, 1 ∨ 2 ∨ 6 ∨ 7 ∨ 8, ...
- The area must have at least n traps, which means in every combination of $(u - n + 1)$ cells in the area, there is at least 1 trap.
  - *CNF clauses:* 1 ∨ 2 ∨ 3 ∨ 4 ∨ 6 ∨ 7, 1 ∨ 2 ∨ 4 ∨ 6 ∨ 7 ∨ 8, ...

Noticing that, if the number of surrounding unassigned cells equals to the cell's value, all of that cells will be automatically marked as Trap.



Moreover, to lessen the complexity of the CNF, if the cell is labeled with 0, the surrounding cells will automatically be labeled as no trap here
(and will eventually be marked as Gem).



2. *Generating CNF automatically*

The model is represented by the flatten version of the map, with each variable assigned either a negative or a positive number (false or true). Therefore, the model cannot contain 0 because it does not differ between positive and negative.

Applying the above principles, the clauses are generated by looping through all the known cells. Moreover, in some cases, some generated clauses are the same, which can increase the size of the formula. Therefore, removing duplicates may seem unnecessary but can partially minimize the complexity.

3. *Use PySAT to solve the CNF*

By using the PySAT module, which is optimized for solving these problems, simply add the generated CNF into the solver. The output model will be either None, which refers to unsatisfiable problem, or the list of positive and negative elements (is a trap or is not a trap correspondingly).

4. *Use DPLL to solve the CNF*

The DPLL (Davis – Putnam – Logemann – Loveland) algorithm works by recursively searching for an assignment that satisfies the formula.

In every instance:

- Simplifying the formula through the use of pure-literal-elimination and unit propagation.
- Choose a random variable and assign a value to it.
- Then run again until no more clauses are in the formula.
  If the algorithm found a contradiction, it backtracks to the nearest assignment and re-assigns the opposite to that variable.

a. Constraints Propagation

The process is to simplify the formula by removing the constraints inside the CNF clauses that go against the proved variable.

E.g.: Supposing we have the clauses in the CNF:

1. $[a]$
2. $[a, b]$
3. $[-a, b, c]$
4. $[c, d, e]$

Choosing the a as the unit to propagate, the CNF will be reduced to:

1. _: this is a *unit clause* that we as a unit to perform propagation.
2. _: already satisfied, no necessary to further considering it.
3. $[b, c]$: the literal $-a$ is pruned because it goes against the considering unit.
4. $[c, d, e]$: remains the same due to having no relation with a.

b. Pure literal elimination

First, we have to understand what a pure literal is. A pure literal is a literal that appears with the same polarity throughout the formula (either true or false, but not both).

E.g.: CNF: $[[a, b, -c], [-b, -c], [a]]$*:* $a, -c$ *are pure literals while* b *is not.*

Then, for every pure literal found, update the formula by using constraint propagation with that literal as a unit.

c. *Unit propagation*

Unit clauses are the clauses that contain only one variable inside it. E.g.: $[a], [-b], \ldots$

While the number of unit clauses is not empty, we will use the variable of that unit clause to update the formula by constraint propagating. Then the unit clauses will be re-checked.

E.g.

1st loop:	2nd loop:	3rd loop:
$[-a, -b, c, d]$	$[-b, c, d]$	$[c, d]$
$[-a, b]$	$[b]$	
$[a]$		

d. *Decide the literal for trying assigning value*

In theory, a value will be assigned at random to a literal (the algorithm reverts to the opposite value if no solution is found). But depending on this, the performance will also be random.

Thus, selecting the literal that appears the most frequently in the current CNF is an additional application strategy to think about.

e. *Pseudocode*

```
DPLL(formula, assigned) → solution(list of assignments or None):
    formula, assignment1 = pure_literal_elimination(formula)
    formula, assignment2 = unit_propagation(formula)
    update assigned with assignment1 and assignment2
    if formula is empty:     return assigned
    if formula contains an empty clause:     return None
    l = choose_literal(formula) # Choose randomly or some other method of evaluation
    assigned[l] = True
    solution = DPLL(formula, assigned)
    if solution is None: # If the assignment is wrong, re-assign
        assigned[l] = False
        solution = DPLL(formula, assigned)
    return solution
```

5. *Use Brute Force*

The brute force method is implemented by generating all possible cases that can happen on the path of solving.

By assigning each value either true or false step by step, eventually it will reach the point where all variables are assigned. Only then will the current state be checked to see whether it meets the CNF or not. If it is, then simply return that case. Otherwise, the next state will be checked until it finds one.

Furthermore, the implementation avoids the assigned cells and all the cells that are already deduced during the CNF-generating steps. By doing this, the number of variables to be checked will be reduced relatively.

6. *Genetic Algorithm (GA)*

Being inspired by the process of natural selection in biology, the Genetic Algorithm method maintains a population of states with a fixed size, as well as forms a fitness function which is used for calculating the fitness of each state. From the old population, we generate a new population in [X/2] iterations, which X is the size of the old population. In each iteration, we choose randomly two states from the old population (the better state's fitness is, the higher state's chosen probability is). These two states will be in process of recombination (crossover) and mutation before being added into the new population.

a. *The representation of each state*

In our solution, we represent each state in the population by the class **Node**, which combines three things:

- The current state of the node, which is the list of integers of where:
  - The positive integer denotes there is a trap at the corresponding position.
  - The negative integer denotes there is no trap at the corresponding position.
  - All integers in the list will correspond unknown cells in the game map, which are not in the list of assigned and inferred cells (*will be presented below*)
- The list of CNF clauses, which serves the calculation of the heuristic value of each value.
- The list of assigned cells and all the cells that are already deduced during the CNF-generating steps, which is used to combine with the current state to form a **complete assignment** of the problem for calculating the heuristic of the node.

About the heuristic value of each node, we calculate it by counting the number of clauses that conflict with the **complete assignment**. We identify that

"A clause is conflict with the assignment if and only if every literal in the assignment is not in the clause."

Prove:

Assume that the state is [A_1, A_2, ..., A_n], it means in the new knowledge base will be added n different clauses corresponding n literal(s) A_1, A_2, ..., A_n in the state.

In the forward direction, when a conflict occurs, it means there exists a clause $[-A_{i_1}, -A_{i_2}, \ldots, -A_{i_k}]$ comprising k negated literal(s) in the knowledge base (Note: A clause [A,B] in the knowledge base is equivalent to A v B, the operator "-" means negation). So, it can be observed that each literal in the state is not in the clause

In the backward direction, when every literal in the state is not in the clause: Because each literal in each clause of the knowledge base is only one of literals in [A_1, A_2, ..., A_n, -A_1, -A_2, ..., -A_n]. So, because every literal in the state is not in the clause, it means the clause is $[-A_{i_1}, -A_{i_2}, \ldots, -A_{i_k}]$ where i_1, i_2, ..., i_k are arbitrary indexes of literals in the state. Because $A_{i_1} \wedge A_{i_2} \wedge \ldots \wedge A_{i_k} \wedge (-A_{i_1} \vee -A_{i_2} \vee \ldots \vee -A_{i_k})$ is always false, it means the clause is conflict with the state

Q.E.D

The current state of the node will be the solution if the heuristic value is 0.

b. *Generate population*

This is the first step of Genetic Algorithm when a population will be instantiated. With a given size of population, in each iteration, a state will be generated randomly as a random assignment for uninferred cells in the game map, then it will be packaged in a node along with the set of CNF clauses and the list of inferred literals.

c. *Fitness function*

With each node, because the heuristic value is the number of clauses that conflict with the **complete assignment**, so the fitness value of each node will be the difference between the total number of CNF clauses and the heuristic value.

d. *Select pair of states*

With each node the old population, we will store the fitness value of each node in a list, then based on this list, we choose randomly two nodes from the old population satisfies the property: *the better state's fitness is, the higher state's chosen probability is*.

e. *Cross over*

Crossing over is the process that we will choose randomly one point in the range of state size, then:

- Each state will be split into two parts at that point.
- Two second parts of two states will be swapped each other and two offstrings will be formed.

f. *Mutation*

After being crossed over, two offsprings will experience a process called mutation, where each offspring will have a chance to negate randomly a literal in the state of the node with a small probability (0.2 in default with our solution).

V. Algorithm Comparisons

Time (measure in nanoseconds)				
Maps	***PySAT***	***DPLL***	***GA***	***Brute Force***
map4.txt	0	994,500	0	0
map5.txt	0	0	524,812,400	112,705,000
map6.txt	988,700	0	68,405,000	1,015,100
map7.txt	988,500	0	0	996,800
map8.txt	0	1,523,200	148,437,399,800	126,484,756,400
map10.txt	461,500	6,993,800	*Intractable*	*Intractable*
map15.txt	998,500	11,023,100	*Intractable*	*Intractable*

In certain ways, backtracking served as the basis for SAT solvers. As a result, we decided that the best algorithm to solve the CNF was to use **DPLL**. So, to ensure that there are at least 2 algorithms which were used to compare with my chosen best algorithm on the aspect of the running time, we decide to implement **GA (Genetic Algorithm)** as a supplementation.

Due to the exponential growth in state space, brute force and other straight-forward algorithms soon reach the maximum limit depth or run out of space. Hence, the measurement will be marked as "intractable".

Comparisons:

- **PySAT** is a high-level library for solving SAT problems; needless to say, it is unparalleled when compared with the other algorithms.
- **DPLL** seems to have solved the problem with ease. This technique can be efficient for minor problem instances, but its performance may degrade in more complex instances.
- The **brute force** approach tests all possible combinations of variable assignments. While conceptually simple, the computations can become infeasible for larger problem instances due to the growth in state space.
- Inspired by the processes of natural selection and evolution, **GA**'s solutions are to evolve over generations. Though the crossover and mutation steps are random, the final solution may take quite a while. Moreover, GAs do not guarantee finding the optimal solution.

VI. Reference

- PySAT

SAT solver's API. *https://pysathq.github.io/docs/html/api/solvers.html*

- GA

Solving Hard Problems.
https://www.mimuw.edu.pl/~erykk/algods/lecture11.html?fbclid=IwAR0UTW9EJu3P0j38ZvDZxznwgW0jPNoi178ScleB2pwYP7BXMbZrAf2-v-s

(2021). Kie Codes. Genetic Algorithms Explained By Example.
https://youtu.be/uQj5UNhCPuo?si=TQfbmzIHCHq5WtKP

(2021). Kie Codes. Genetic Algorithm from Scratch in Python (tutorial with code).
https://youtu.be/nhT56blfRpE?si=G_E1MTUnmfGfcLpX

- DPLL

(2018). marcmelis. dpll-sat.

https://github.com/marcmelis/dpll-sat/blob/master/solvers/original_dpll.py