HO CHI MINH NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

---o0o---

**Introduction to Artificial Intelligent**

**A blue and white logo

Description automatically generated**

**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 2](#_Toc164183052)

[a. Team information 2](#_Toc164183053)

[b. Task division 2](#_Toc164183054)

[II. Environment Requirement 2](#_Toc164183055)

[III. Algorithms 3](#_Toc164183056)

[1. Logical principles in generating CNF 3](#_Toc164183057)

[2. Use PySAT to solve the CNF 4](#_Toc164183058)

[3. Use DPLL to solve the CNF 4](#_Toc164183059)

[a. Constraints Propagation 4](#_Toc164183060)

[b. Pure literal elimination 5](#_Toc164183061)

[c. Unit propagation 5](#_Toc164183062)

[d. Pseudocode 5](#_Toc164183063)

[4. Use Brute Force 6](#_Toc164183064)

[5. Genetic Algorithm (GA) 6](#_Toc164183065)

[IV. Algorithm Comparisons 6](#_Toc164183066)

# Assignment Planner

### Team information

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

### 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 |

### 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% |

# Environment Requirement

*Python version:* 3.10+.

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

*Usage:* 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.

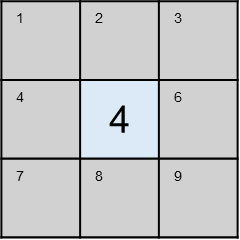
# Algorithms

## 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.

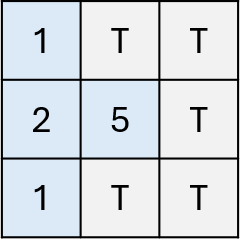
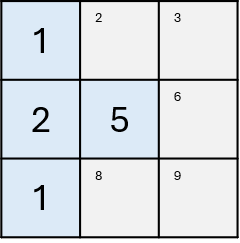


*E.g.: There is traps in 4 of the remaining cells (1, 2, 3, 4, 6, 7, 8, 9)*  
*Assuming the trap cells are 1, 2, 3, 4, the CNF clause we achieve is .*

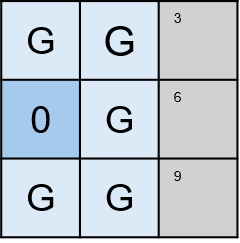
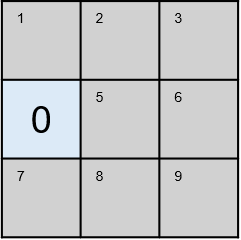
Therefore, we can infer two more principles (supposing the cell's value is and the number of surrounding unassigned cells is ):

* The area must have at most n traps. In other words, considering every combination of cells in the area, it is guaranteed to have at least one   
  safe cell (gem).
* *CNF clauses: , , , …*
* The area must have at least n traps, which means in every combination   
  of cells in the area, there is at least 1 trap.
* *CNF clauses: , , …*

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).



## 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.

## 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).

## 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.

### 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:*

Choosing the 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. : the literal is pruned because it goes against the considering unit.
4. : remains the same due to having no relation with .

### 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: : are pure literals while is not.*

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

### Unit propagation

Unit clauses are the clauses that contain only one variable inside it. E.g.:

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:

### 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

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

## 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.

## Backtracking

## Genetic Algorithm (GA)

# Algorithm Comparisons

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | Time (Measure in nanoseconds) | | | | |
| *Maps* | ***PySAT*** | | ***DPLL*** | ***GA*** | ***Brute Force*** | ***Backtrack*** |
| map4.txt | 0 | | 994,500 | 0 | 0 | 0 |
| map5.txt | 0 | | 0 | 524,812,400 | 112,705,000 | 0 |
| map6.txt | 988,700 | | 0 | 68,405,000 | 1,015,100 | 0 |
| map7.txt | 988,500 | | 0 | 0 | 996,800 | 0 |
| map8.txt | 0 | | 1,523,200 | 148,437,399,800 | 126,484,756,400 | 2,015,800 |
| map10.txt | 461,500 | | 6,993,800 | *Intractable* | *Intractable* | 20,610,800 |
| map15.txt | 998,500 | | 11,023,100 | *Intractable* | *Intractable* | 8,547,300 |

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. Furthermore, we provide a backtracking version, in which no phase of simplifying clauses is included – only the backtrack steps are recorded.

It is worth noticing that, in DPLL, there are steps to be chosen randomly; therefore, the result may vary between runs.

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.
* The **backtracking** method involves assigning a variable until reaching a valid solution. It prunes the search space by abandoning the partial solutions that are insatisfiable.
* 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.