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**

No table of contents entries found.

# Assignment Planner

# 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.

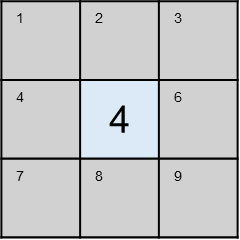
# Algorithms

## 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.

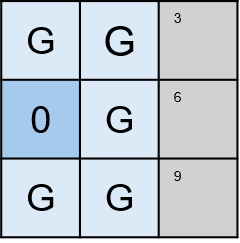
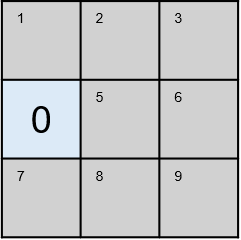


*E.g.: There is traps in 4 of the remaining cells (1, 2, 3, 4, 6, 7, 8, 9)*  
*Assuming the trap cells are 1, 2, 3, 4, the CNF clause we achieve is .*

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: , , …*

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).



## 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 .

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