Vietnam National University Ho Chi Minh City

University of Science

Faculty of Information Technology

PROJECT 2:
Gem Hunter

Course	Introduction to AI
Class	22CLC02
Students	**22127357 – Phạm Trần Yến Quyên** **22127402 – Bế Lã Anh Thư** **22127459 – Phạm Thanh Vinh** **22127488 – Trương Thanh Toàn**
Github:	AI-Project2
Overleaf:	AI-Project2 Report

HCMC, 2024

To have a good experience, please use a PDF reader that supports hyperlinks

Mục lục

1 Definitions/Basic Concepts:

1.1 CNF:

The conjunctive normal form is a way of expressing a formula in the boolean logic. It states that a formula is in CNF if it is a conjunction of one or more than one clause, where each clause is a disjunction of literals. In other words, it is a product of sums where $\wedge$ symbols occur between the clauses and $\vee$ symbols occur in the clauses.

1.2 SAT Problem

The Boolean satisfiability problem (SAT) is a fundamental problem in computer science and mathematics. It's concerned with determining whether there exists an assignment of truth values (`true`|`false`) to a set of Boolean variables that satisfies a given Boolean formula.
The problem contains the following notations:

1. Boolean Formula: The input to the SAT problem is typically a Boolean formula. This formula is composed of Boolean variables (which can take values of true or false), logical connectives (like AND, OR, NOT), and parentheses for grouping:

$$(X_2 \vee X_5) \wedge (X_2 \vee X_6) \wedge (X_5 \vee X_6) \wedge (\neg X_2 \vee \neg X_5 \vee \neg X_6)$$

2. Satisfiability: A satisfying assignment for a Boolean formula is an assignment of truth values to its variables that makes the formula evaluate to true. If there exists at least one such assignment, the formula is said to be satisfiable.

3. Unsatisfiability: If there's no assignment that makes the formula evaluate to true, the formula is unsatisfiable.

4. Decision Problem: The SAT problem is a decision problem, meaning the question it asks is a yes-or-no question: "Is there a satisfying assignment for this Boolean formula?"

EXTRA: One of the key reasons the SAT problem is so important is its computational complexity. While the problem itself is in NP (nondeterministic polynomial time), it's one of the first problems proven to be NP-complete (*proved by Stephen Cook at the University of Toronto in 1971*). This means that if there exists a polynomial-time algorithm for solving the SAT problem, then there exists a polynomial-time algorithm for solving all problems in NP, which would imply P = NP—a major unsolved question in computer science.

2 GEM HUNTER

– Gem Hunter is a strategic puzzle game where players explore a grid to reveal valuable gems while avoiding dangerous traps. The objective is to reveal all the gems on the grid without triggering any traps. Players must use logic and deduction to strategically uncover tiles and mark potential trap locations. You will know the location of the trap through the tiles on the map. Each number on a tile represents the number of traps around that tile.

– The rules are as follows:

- The grid/board has a dimension of `NxN` (A square grid).
- A trap is represented by the symbol `T`.
- A gem is represented by the symbol `G`.
- Each empty cell (cell that has a number in it) contains a number from 0-9 that represents the number of traps around that cell.
- An unkown cell is represented by the symbol `_` (can be `T | G`).
- A cell can have at most 8 neighbors. (up, down, left, right, and diagonal)
- The objective is to determine the location of the traps and the gems.

```
File format:
            Input:              |       Output:
            3, _, 2, _          |       3, T, 2, G
            _, _, 2, _          |       T, T, 2, G
            _, 3, 1, _          |       T, 3, 1, G
```

– Due to the nature of this Game being a SAT Problem, the Game board is to be formatted into a `CNF`. In order to do this, preparations are made.

3 Preparation

3.1 Clauses conventions

To put it simply, we will have some conventions. First, call the cell that has the number $N_{i,j}$. i is the order of rows and j is the order of columns. For example, if $N_{3,4} = 5$, it means that the cell at row 3, column 4 contains number 5.
Next, if a cell doesn't contain a number, it can be a trap or gem, call that cell $T_{i,j}$. If $T_{i,j} = True$, it means that the cell at row i, column j contains a trap, if $T_{i,j} = False$, it contains a gem.
To write constraints for cells containing numbers, variables for nearby cells need to be defined. At cell $N_{i,j}$, there are 8 nearby cells:

$T_1 = T_{i-1,j}$	$T_5 = T_{i-1,j-1}$
$T_2 = T_{i+1,j}$	$T_6 = T_{i-1,j+1}$
$T_3 = T_{i,j-1}$	$T_7 = T_{i+1,j-1}$
$T_4 = T_{i,j+1}$	$T_8 = T_{i+1,j+1}$

3.2 Constraints for cells containing numbers

- $N_{i,j} = 1$

  $X_1 = T_1 \vee T_2 \vee T_3 \vee T_4 \vee T_5 \vee T_6 \vee T_7 \vee T_8$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2}$, for all combinations of integer i_1, i_2 such that $1 \leqslant i_1 < i_2 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$.

- $N_{i,j} = 2$

  $X_1 = T_{i_1} \vee T_{i_2} \vee T_{i_3} \vee T_{i_4} \vee T_{i_5} \vee T_{i_6} \vee T_{i_7}$, for all combinations of integer $i_1, i_2, i_3, i_4, i_5, i_6, i_7$ such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 < i_6 < i_7 \leqslant 8$. All combinations are connected by $\wedge$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2} \vee \neg T_{i_3}$, for all combinations of integer i_1, i_2, i_3 such that $1 \leqslant i_1 < i_2 < i_3 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$

- $N_{i,j} = 3$

  $X_1 = T_{i_1} \vee T_{i_2} \vee T_{i_3} \vee T_{i_4} \vee T_{i_5} \vee T_{i_6}$, for all combinations of integer $i_1, i_2, i_3, i_4, i_5, i_6$ such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 < i_6 \leqslant 8$. All combinations are connected by $\wedge$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2} \vee \neg T_{i_3} \vee \neg T_{i_4}$, for all combinations of integer i_1, i_2, i_3, i_4 such that $1 \leqslant i_1 < i_2 < i_3 < i_4 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$.

- $N_{i,j} = 4$

  $X_1 = T_{i_1} \vee T_{i_2} \vee T_{i_3} \vee T_{i_4} \vee T_{i_5}$, for all combinations of integer i_1, i_2, i_3, i_4, i_5 such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 \leqslant 8$. All combinations are connected by $\wedge$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2} \vee \neg T_{i_3} \vee \neg T_{i_4} \vee \neg T_{i_5}$, for all combinations of integer i_1, i_2, i_3, i_4, i_5 such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$.

- $N_{i,j} = 5$

  $X_1 = T_{i_1} \vee T_{i_2} \vee T_{i_3} \vee T_{i_4}$, for all combinations of integer i_1, i_2, i_3, i_4 such that $1 \leqslant i_1 < i_2 < i_3 < i_4 \leqslant 8$. All combinations are connected by $\wedge$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2} \vee \neg T_{i_3} \vee \neg T_{i_4} \vee \neg T_{i_5} \vee \neg T_{i_6}$, for all combinations of integer $i_1, i_2, i_3, i_4, i_5, i_6$ such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 < i_6 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$.

- $N_{i,j} = 6$

  $X_1 = T_{i_1} \vee T_{i_2} \vee T_{i_3}$, for all combinations of integer i_1, i_2, i_3 such that $1 \leqslant i_1 < i_2 < i_3 \leqslant 8$. All combinations are connected by $\wedge$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2} \vee \neg T_{i_3} \vee \neg T_{i_4} \vee \neg T_{i_5} \vee \neg T_{i_6} \vee \neg T_{i_7}$, for all combinations of integer $i_1, i_2, i_3, i_4, i_5, i_6, i_7$ such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 < i_6 < i_7 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$.

- $N_{i,j} = 7$

  $X_1 = T_{i_1} \vee T_{i_2}$, for all combinations of integer i_1, i_2 such that $1 \leqslant i_1 < i_2 \leqslant 8$. All combinations are connected by $\wedge$.

  $X_2 = \neg T_{i_1} \vee \neg T_{i_2} \vee \neg T_{i_3} \vee \neg T_{i_4} \vee \neg T_{i_5} \vee \neg T_{i_6} \vee \neg T_{i_7} \vee \neg T_{i_8}$, for all combinations of integer $i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8$ such that $1 \leqslant i_1 < i_2 < i_3 < i_4 < i_5 < i_6 < i_7 < i_8 \leqslant 8$. All combinations are connected by $\wedge$.

  $Y = X_1 \wedge X_2$.

- $N_{i,j} = 8$

  $Y = T_1 \wedge T_2 \wedge T_3 \wedge T_4 \wedge T_5 \wedge T_6 \wedge T_7 \wedge T_8$.

3.3 Proofs

If $N_{i,j} = k$, then the number of traps around that cell must be k. Create two clauses X_1 and X_2, X_1 is true if the number of traps around cells i, j is equal to or greater than k, X_2 is true if the number of traps around the cells i, j is equal to or lower than k.
Create a new clause $Y = X_1 \wedge X_2$. Y is only true if X_1 and X_2 are true, which means number of traps is k.

- Constraints for X_1

  The number of traps must be equal to or greater than k, or the number of cells that are not a trap is lower than $8 - k$. Create all $8 - k + 1$ cell combinations of nearby cells, and check if in those combinations, is there at least one cell is a trap. If a combination is false, it means that at least $8 - k + 1$ cells are not a trap, and X_1 will be false.

  The equation is: $X_1 = \forall i_1, i_2, ..., i_{8-k}, i_{8-k+1}, 1 \leqslant i_1 < i_2 < ... < i_{8-k} < i_{8-k+1} \leqslant 8 \bigwedge (T_{i_1} \vee T_{i_2} \vee ... \vee T_{i_{8-k}} \vee T_{i_{8-k+1}})$

- Constraints for X_2

  The number of traps must be equal to or lower than k. Create combinations of nearby cells to check if the number of traps in nearby cells is greater than k, each combination contains $k + 1$ cells. If a combination is false, it means that there are at least $k + 1$ cells that contain traps, which leads to X_2 being false.

  The equations is: $X_2 = \forall i_1, i_2, ..., i_k, i_{k+1}, 1 \leqslant i_1 < i_2 < ... < i_k < i_{k+1} \leqslant 8 \bigwedge (\neg T_{i_1} \vee \neg T_{i_2} \vee ... \vee \neg T_{i_k} \vee \neg T_{i_{k+1}})$

- $Y = X_1 \wedge X_2$

  The number of traps of nearby cells must be equal to k for Y to be true.

4 Implementation Methods

4.1 Automatic CNFs Generation

The CNF representation is a crucial step for applying logical algorithms like SAT solvers to solve the game or reason about it.
To write a method that can create CNFs automatically, a new class called `BoardCNF` is created. This class will have methods to save the game board, gen clauses for each cell that contains numbers, and gen clauses for the whole board. `BoardCNF` also saves the ID number of each cell so that the clauses can be generated easily. Now, consider all methods:

- `def gen_clauses(self) -> list:` This method will iterate the whole board and check if any cells contain numbers. If a cell contains numbers, it will call *method add_ cells_ clauses* to generate new clauses based on that cell. After finishing iterating, this method will return a list containing all clauses.

- `def add_cells_clauses(self, row: int, col: int) -> None:` This method will count the number of possible traps in nearby cells, normally, it would be 8, but in some cases, the cell can be at the corner of the board, so the number of possible traps in nearby cells must be counted. After that, *gen_ combine* will be called.

- `def gen_combine(num_trap_cells: int, pos_trap_cells: list) -> list:` This method will generate all the combinations of possible cells and add them to the clauses. The return value is a list.

4.2 Brute-Force & Backtracking

4.2.1 Brute-Force

– Initialization:

  - `gen_board(filePath)`: read the board from an input file and store it internally, each cell in the board can be "_ empty cell or be a numbered cell indicating the number of traps adjacent to it.
  - `has_number_neighbor(i, j)`: this function will check does in the neighborhood of `cell[i][j]` have any numbered cell or not, if there exists, return True, otherwise return `False`. This function traverse through all the neighbor by generate Cartesian product of `[-1,0,1]` (Ex: `[-1,-1]` is top left, `[-1,1]` is top right and so on).

– Solution Generation: The `brute_force_solve` method exhaustively generates all possible combinations of traps (T) and gems (G) for the empty cells on the board. It utilizes `itertools.product` to create the Cartesian product of possible trap and gem assignments. The generated solutions are then validated using the `is_valid_solution` method.

– Validation: The `is_valid_solution` method iterates over each cell on the board. For each cell containing a number, it checks whether the number of adjacent traps satisfies the number indicated in the cell. This validation process involves examining neighboring cells, akin to generating clauses in Conjunctive Normal Form (CNF), where each clause represents a condition that must be satisfied.

– Solution Refinement: Following solution generation, a refinement step is performed to ensure the correctness of trap assignments. This step iterates over the generated solution board, utilizing `has_number_neighbor` to verify the correctness of trap assignments and modify any incorrect trap placements.

– Execution: The run method orchestrates the generation and validation process, invoking the necessary functions to generate the game board, execute the algorithm, and return the solution.

4.2.2 Backtracking

– Initialization and Solution Generation: The Backtracking class shares initialization and board generation methods with the BruteForce class.

– Constraint Satisfaction: The backtrack method recursively explores the solution space, guided by constraint satisfaction principles. It starts by checking if a valid solution has been found or if there are unassigned cells remaining. Next, it selects the next unassigned cell with the fewest remaining options, prioritizing cells with the least number of adjacent numbered cells. For each option (trap or gem), it recursively explores the solution space, checking validity at each step using the `valid_solution` method. If a dead-end is reached, the algorithm backtracks, undoing the current assignment and exploring alternative options.

– Solution Validation: Similar to the BruteForce algorithm, the Backtracking algorithm validates solutions by ensuring that the number of traps adjacent to each numbered cell matches the indicated number and that the total number of traps in the neighborhood of each numbered cell does not exceed the indicated number.

4.3 PySAT Library

4.3.1 Justification

– The implementation uses the PySAT library to encode the variables and constraints of the Gem Hunter puzzle as a CNF formula. It then uses a SAT solver to find a satisfying assignment that represents the solution to the puzzle. The implementation is efficient and scalable, as it can handle grids of any size and solve the puzzle in a reasonable amount of time.

– The use of a SAT solver ensures that the solution is correct and optimal, as it guarantees that all constraints are satisfied.

4.3.2 Solvers

The PySAT library comprises of many Solvers (AKA algorithmic tools) to solve CNF. Here are 3 of the most common ones:

- MiniSAT (2.2 version): Minisat22 is a lightweight, efficient, and widely used SAT solver. It implements modern SAT-solving techniques, including conflict-driven clause learning (CDCL). Some quirks:

    1. Clause Learning: Minisat22 uses a conflict-driven clause learning (CDCL - See more at **Section 4.4**) algorithm. When it encounters a conflict during the search, it analyzes the conflict to learn new clauses, which are then added to the formula to prevent similar conflicts in the future.
    2. Backtracking and Restarting: It uses backtracking to explore the search space and backtrack when a conflict is encountered. Minisat22 also employs restart strategies, which periodically reset the search to explore different parts of the search space.
    3. Watched Literals: Minisat22 uses a technique called watched literals to efficiently propagate truth assignments. It monitors clauses by watching literals, ensuring that whenever a watched literal becomes true, the clause is satisfied without further computation.
    4. Conflict Analysis: When a conflict is detected, Minisat22 analyzes it to identify the cause of the conflict and learns a new clause based on this analysis. This learned clause helps in avoiding similar conflicts in subsequent iterations.

- Glucose (A play on the words Glue and Clauses): Glucose is heavily based on Minisat and enhances on it, focusing on performance improvements and advanced algorithms. Therefore also uses the CDCL Method of Solving SAT
  Some quirks:

  1. Enhanced Clause Learning: Glucose builds upon Minisat's CDCL algorithm with enhanced clause learning techniques. It learns more effective clauses during conflict analysis, leading to faster convergence.
  2. Dynamic Restart Strategies: Glucose employs dynamic restart strategies, adjusting the restart frequency based on the progress of the search. This helps in adapting to different problem instances and improving overall performance.
  3. Phase-Saving Heuristics: Glucose introduces phase-saving heuristics, which remember the decision variable polarity across restarts. This helps in guiding the search towards more promising parts of the search space.
  4. Clause Database Management: Glucose optimizes the management of the clause database to reduce memory consumption and improve cache efficiency, thereby speeding up the solving process.

- CaDiCaL (Compact and Distributed Cache-friendly Learned): It's designed to be lightweight and cache-friendly, which is particularly beneficial for large-scale SAT instances. And because of that, it can handle very large problem instances efficiently, making it a popular choice for solving challenging SAT instances.
  Some quirks:

  1. Cache-Friendly Data Structures: CaDiCaL is designed with cache-friendly data structures to optimize memory access patterns, which is crucial for performance, especially on modern computer architectures.
  2. Parallel Search: It supports parallel search, allowing multiple threads to work on different parts of the search space simultaneously, thus speeding up the solving process on multi-core systems.
  3. Clause Elimination: CaDiCaL employs clause elimination techniques to reduce the size of the formula dynamically. This helps in simplifying the problem and improving the efficiency of the solver.
  4. Clause Learning and Conflict Analysis: Similar to Minisat and Glucose, CaDiCaL uses clause learning and conflict analysis to guide the search process efficiently. It learns from conflicts and adjusts its search strategy accordingly.

4.3.3 Implementation:

A `PySatSolver` is created to establish all the necessary tools for solving the input Game board.

```
class PySatSolver:
    def __init__(self, clauses: list, solver: str):
        cnf = CNF(from_clauses=clauses)
        self.solver = Solver(name=solver, bootstrap_with=cnf)

    def solve(self) -> list | None:
        result = self.solver.solve()
        if result:
```

```
            return self.solver.get_model()
        return None
```

Where:

- Constructor (`__init__`): The constructor initializes a `PySatSolver` object with the provided list of clauses and the chosen SAT solver.
  - Parameters:
    + `clauses(list)`: A list of clauses that represent the SAT problem generated from the previously explained `BoardCNF` class.
    + `solver(str)`: The name of the SAT solver to be used (e.g., g4, g3, m22, etc.).
  - Actions:
    + Converts the list of clauses into CNF using the PySAT's `CNF(from_clauses=clauses)` function (Automatic CNF generation).
    + Initializes the solver with the specified solver name and the CNF formula.
- solve Method: The solve method is responsible for solving the SAT problem.
  - Returns:
    + If a satisfying assignment is found, it returns the assignment as a list.
    + If no satisfying assignment is found, it returns `None`.
  - Steps:
    + Invokes the solver's solve method to attempt to find a satisfying assignment.
    + If a solution is found (result is `True`), it retrieves the satisfying assignment using the solver's `get_model()` method and returns it.
    + If no solution is found, it returns `None`.

4.4 CDCL Algorithm

Similarly to the Tools in PySAT, we've implemented CDCL for solving the Gem Hunter problem due to it efficiency.

4.4.1 About CDCL

- Conflict-Driven Clause Learning (CDCL) SAT solvers are among the most efficient methods for solving Boolean satisfiability problems (SAT). CDCL SAT solvers are primarily inspired by DPLL solvers. In which DPLL corresponds to backtrack search, where each step a variable and a propositional value are selected for branching purposes. With each branching step, two values can be assigned to a variable: `0 | 1`. Branching corresponds to assign the chosen value to the chosen variable. Afterwards, the logical consequences of each branching step are evaluated. Each time an unsatisfied clause (conflict) is identified, backtracking is executed.
- Key features of CDCL SAT solvers include:

- Conflict-Driven: The solver makes decisions and resolves conflicts based on a conflict analysis mechanism. When a conflict occurs, it identifies the cause of the conflict and uses that information to guide future decisions.
- Clause Learning: CDCL solvers dynamically learn from conflicts by adding learned clauses to the formula. These learned clauses help to prune the search space, making the solver more efficient over time.
- Backtracking: When a conflict is encountered, the solver backtracks to a previous decision level and makes a different choice. This process continues until a satisfying assignment is found or all possibilities have been explored.
- Efficiency: CDCL solvers are highly efficient and can handle large, real-world problem instances. Their performance has made them the method of choice for many SAT-solving applications.

4.4.2 Preliminaries

In this section, a provision for the formal overview of the concepts and notations used in the analysis is given as folllow:

- Variables and Formulas: Let $X = \{x_1, x_2, \ldots, x_n\}$ represent the set of variables in a Conjunctive Normal Form (CNF) formula F. A CNF formula consists of clauses connected by conjunctions ($\wedge$).
- Clause Structure:
  - Each clause ω is composed of literals, which can be connected by disjunctions ($\vee$).
  - A literal represents a variable (x) or its negation ($\neg x$).
- Characteristics of a clause:
  - `Unsatisfied`: all literals evaluate to 0 or the clause is empty.
  - `Satisfied`: at least one literal evaluates to 1.
  - `Unit`: only one literal is unassigned while others are assigned to 0.
  - `Unresolved`: does not fall into the above three categories.
- Notation:
  - F: CNF formula.
  - ω: Clause.
  - l: Literal.
  - ν: set of assignments
- Data Structures:
  - Literal, Clause, CNF: Representations of CNF formulas.
  - Assignment: Contains information about a variable assignment, including variable, value, antecedent, and decision level.

  - Assignment storage: A dictionary mapping variables to their corresponding Assignment objects (inherits Python dict).

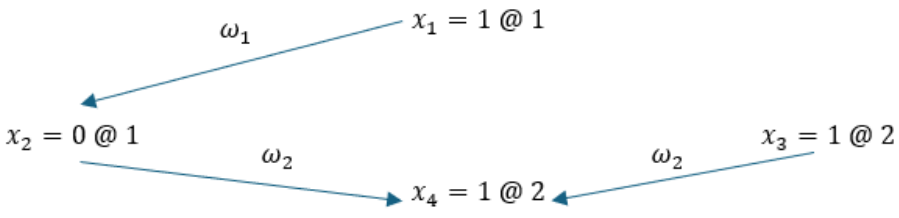
- Assignment and *Antecedent*:
  - An assignment ν maps variables to boolean values or "unassigned".
  - A unit clause implies the value of its last unassigned literal, which becomes the *antecedent* of that assignment.

- Basic Idea:
  - CDCL randomly chooses unassigned variables and assigns a boolean value, akin to **DFS** (Depth-first tree-search).
  - And because CDCL employs a **DFS Algorithm**, the following steps are performed:
    * At each node, two assignments are possible.
    * The level of a node in the tree corresponds to the *decision level*.
    * Backtracking occurs upon encountering conflicts.

- Conflict Handling:
  - When encountering a conflict, DPLL backtracks to the parent node and selects alternative assignments.
  - Unlike DPLL, CDCL uses non-chronological backtracking, aiming to skip more steps.
  - The process of determining a backtrack level involves `conflict_analysis`.

- Variable Assignment: There are **2 situations** for assigning a value to a variable.
  - Normal random assignment or using a heuristic for efficiency in large knowledge bases (KBs).
  - Implication of the last unassigned literal in a unit clause, where the unit clause becomes the antecedent of the assignment.

- Implication Graph: A directed graph representing assignments and their antecedents is presented as follow:



Where:

  - Vertices represent assignments with their decision levels.
  - Edges link assignments to their antecedents.

- UIP − Unique Implication Point: it is a node at a concrete decision level in the implication graph such that ***all*** paths (starting with the same decision level to UIP) to conflict node pass through it. In the above implication graph, suppose x_4 implies a conflict, then at decision level 2, x_4 and x_2 are UIPs. In this project, to learn new clause infer from KB, we use a commonly-used learning scheme called 1-UIP or First-UIP. You can choose some other learning schemes as desired, such as `Rel_Sat`,...

4.4.3 Algorithm

In the below pseudo-codes, `F` denotes the CNF formula, ν denotes the (partial) set of assignments The Conflict-Driven Clause Learning algorithm is described below:

- Unit Propagation: check existence of unit clauses in `F`, assign the unique unassigned literal to `true`. Keep doing this until there is no unit clause or a conflict occurs. If there is no unit clause and no conflict, our set of assignments is good up till now. If our set of assignments suddenly raises a conflict, there must be one or some assignments *made at current decision level* triggered it; and we're gonna analyze them in `CONFLICT_ANALYSIS`.

Algorithm 1 `UNIT_PROPAGATION`(`F`, ν)

```
Output: returns the conflict clause
finished ← false
while have not finished do:
    // there is potentially at least one unit clause
    finished ← true
    for each clause in F do
        status ← status of current clause (is one of the following: unit, satisfied, unsatisfied, unresolved)
        if clause is unit then:
            l ← the unique unassigned literal
            choose boolean value val for variable var in l such that l is True
            ν ← ν ∪ {var ← val}
            // a new variable is assigned so that there may be a new unit clause, cannot finished here
            finished ← false
        else if clause is unsatisfied then:
            // an assignment we made at current decision level causes a conflict
            return conflict flag and conflict clause
        end if
    end for
end while
return nothing (as there is no conflict)
```

- `PICK_VARIABLE`(`F`, ν): randomly choose an unassigned variable and a boolean value for it. Besides, in lieu of choosing arbitrarily, you can use a heuristic function
- `RESOLUTION_OPERATION`(clause1 ω_1 , clause2 ω_2, conflict_literal l): l is in one clause and $\neg l$ is in another. Then, this operation results in a new clause $\omega = \omega_1 \cup \omega_2 \setminus \{l, \neg l\}$. By constantly invoking this operation, the clause will reduce in size.

- Conflict analysis:
  - The occurence of conflict is due to assignments made at current decision level. Consequently, the clause that implied these assignments (called antecedents) are potentially resolve together to produce a new short clause. Note that, a short clause is a good clause as it rejects more solutions than a longer one.
  - Idea: get list of literals in the conflict clause whose variable is assigned at current decision level. For each literal in the obtained list, we achieve the corresponding antecedent, then invoke resolution operation between the previous intermediate clause (intermediate clause is the clause resulted from a resolution operation, initially be the given conflict clause). Get new list of literals, repeat the above process until we meet the first UIP (*the first UIP is met iff number of literals whose variable is assigned at current decision level is 1*). At that time, the deepest decision level in the new learnt clause is current decision level, and the second deepest is the decision level of first UIP from which you can restart your assigning process. Note that, if the list of decision levels extracted from the new clause has only 1 decision level (which means it does not have the second deepest *dl*), it is a signal that the conflict has conclude our process since there is nowhere to go: decision level is set to 0 (to the root), still allow to assign but conflict is prohibited.
  - Denote ω_i is the intermediate clause when analyzing conflict at step i; l is a literal in ω whose variable was assigned at current decision level; $a \vee b$ means we apply resolution operator on 2 clauses a and b; $\alpha(l)$ denotes the *antecedent* of the assignment of the variable in literal l. The formula of conflict analysis is given as below to give you an overview of this functionality.

$$\omega_i = \begin{cases} \texttt{conflict\_clause}, & \text{if } i = 0 \\ \omega_{i-1} \vee \alpha(l), & \text{for each } l \text{ in } \omega \ (l \text{ is not chosen twice}) \end{cases}$$

  - The final intermediate clause will be the new clause to learn (added to KB).
  - When implement the algorithm, to avoid a literal to be chosen twice, we made use of Python `next()` method when choosing literals.

Algorithm 2 `CONFLICT_ANALYSIS` (`conflict_clause`, ν)

1: Output: returns decision level to backtrack to, and the new clause to learn
2: **if** current decision level is 1 **then**
3: // a conflict occur at root: the CNF is unsatisfied
4: return a negative decision level as a signal
5: **end if**
6: rename variable: `clause` ← `conflict_clause` ▷ the intermediate clause
7: `literals` ← list of literals whose variable was assigned at current decision level.
8: **while** number of literal in `literals` $\neq 1$ **do**
9: choose a literal l that has antecedent (those in the list `literals` whose assignments were implied by a clause in formula `F`, not randomly chosen or chosen with heuristic)
10: `clause` ← result after resolving `clause` with the obtained antecedent
11: reconstruct list `literals` with the same principle as above
12: **end while**
13: `decision_levels` ← list of decision levels of assignments of all literals in the final `clause`
14: **if** there is only 1 element in `decision_levels` **then**
15: return backtrack level 0 and new learnt clause
16: **else**
17: return the second deepest decision level in `decision_levels`, and the new learnt clause
18: **end if**

- `BACKTRACK`(ν, backtrack level b): backtrack to the given decision level b, remove all assignments made at decision level deeper than b.

From all of the above, the below Algorithm steps are infered.

Algorithm 3 CDCL(F, ν)

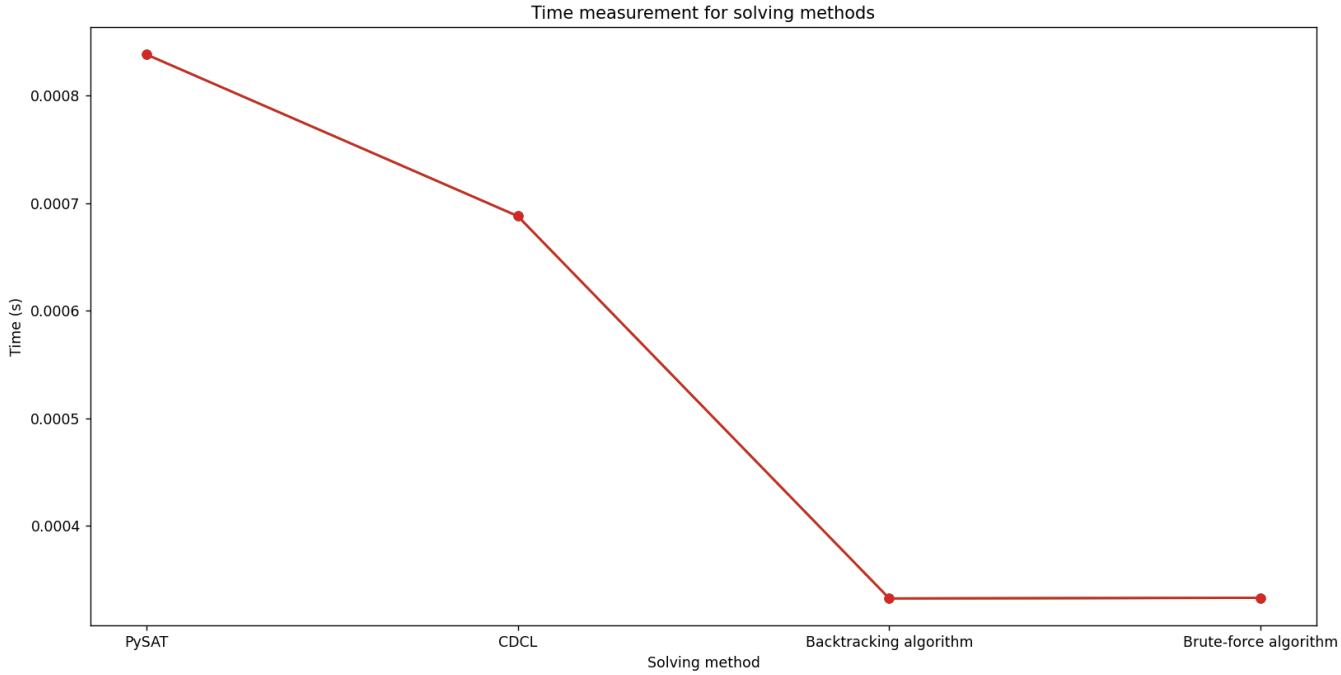
```
Output: returns set of final assignments in which all variables were assigned
if F is initially conflict then
    return NO_SOLUTION
end if
dl ← 0
while there are unassigned variables do
    var, val ← PICK_VARIABLE(F, ν)
    dl ← dl + 1
    ν ← ν ∪ {var = val}
    while 1 do
        conflict_claue ← UNIT_PROPAGATION(F, ν)
        if no conflict_clause then
            break
        end if
        backtrack_level, new_learnt_clause ← CONFLICT_ANALYSIS(conflict_clause, ν)
        if backtrack_level < 0 then
            return NO_SOLUTION
        end if
        add new_learnt_clause to F
        BACKTRACK(ν, backtrack_level)
        dl ← backtrack_level
    end while
end while
return ν
```

In conclusion, the CDCL algorithm offers a systematic approach to solving CNF formulas by efficiently navigating the search space, handling conflicts, and managing assignments. Its utilization of decision levels, backtracking, and conflict analysis contributes to its effectiveness in solving complex logical problems.

5 Comparisions:

The tests are performed on respectively: `3x4/11x11/66x66` Game Boards and ran 3 times each to get the average of 3 time complexity records (PySAT uses Glucose 4.0). This is to fully push the extend of the methods. Here are the results from `'time_measure_test.py'`:

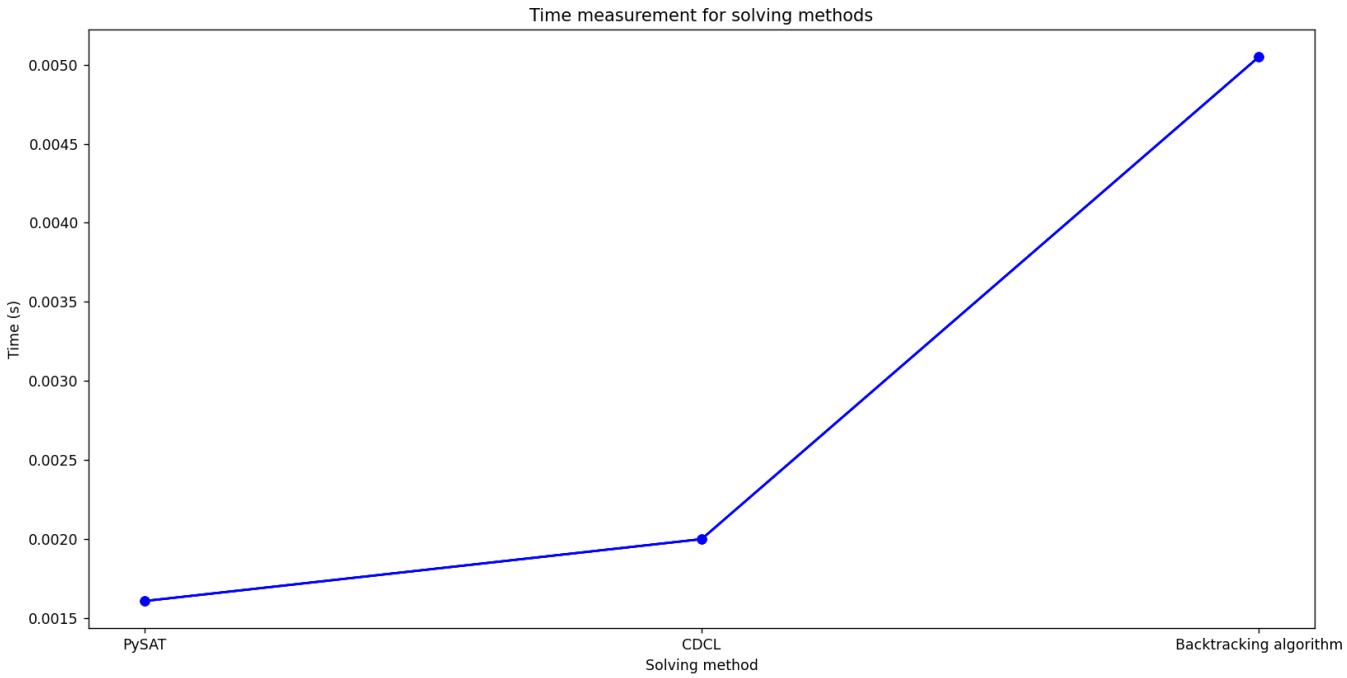
```
3x4:
        PySAT: 0.0008384386698404948 second
        -----------------
        CDCL: 0.0006880760192871094 second
        -----------------
        Backtracking algorithm: 0.0003325939178466797 second
        -----------------
        Brute-force algorithm: 0.0003330707550048828 second
        -----------------
```



3x4 Time Complexity

NOTE: Due to brute-forcing taking too long from this point on it is not included.

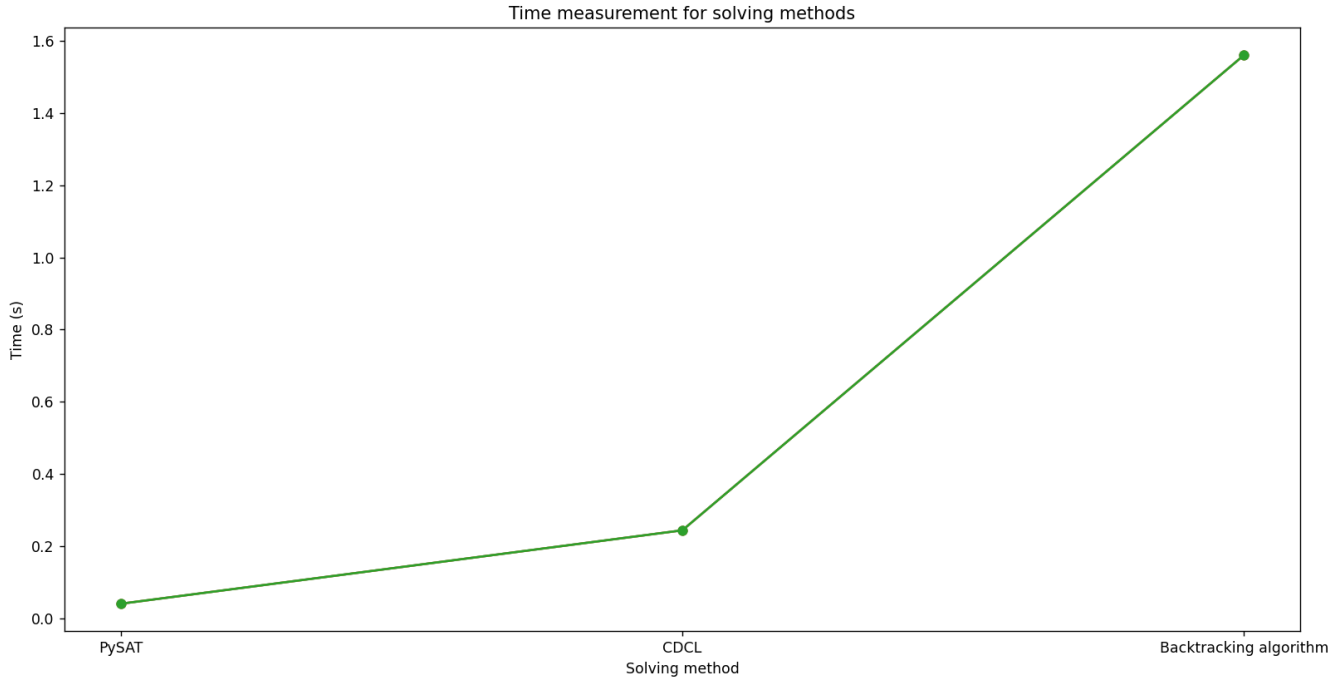
```
11x11:
        PySAT: 0.0011323293050130208 second
        -----------------
        CDCL: 0.0016644795735677083 second
        -----------------
        Backtracking algorithm: 0.0049936771392822266 second
        -----------------
        Brute-force algorithm: ...
        -----------------
```



11x11 Time Complexity

66x66:

```
PySAT: 0.03778 second
-----------------
CDCL: 0.2431 second
-----------------
Backtracking algorithm: 1.5436 second
-----------------
Brute-force algorithm: ...
-----------------
```



66x66 Time Complexity

Comments: The time complexity of the Solvers depends on the size of the CNF formula generated from the Gem Hunter problem (game board in size `m x n`):

- The number of variables in the CNF formula is `n * m`, representing the cells on the board.
- The number of clauses generated from the board is proportional to the number of cells and their neighbors.
- The number of sub-clauses generated for each cell containing numbers is $C_8^{8-N_{ij}+1}$ + $C_8^{N_{ij}+1}$.
- Denote that, E number of empty cells that are not nearby numbered cells and L is the number of numbered cells.

  The total number of clauses generated is $(C_8^{8-N_{ij}+1}$ + $C_8^{N_{ij}+1})$ * L + E.

From that Generation:

– The PySAT solver has an exponential time complexity in the worst case, but it is efficient for solving the Gem Hunter puzzle.

– The time complexity of the CDCL solver is similar to the PySAT solver, as it also generates clauses from the board.

– The backtracking algorithm has an exponential time complexity in the worst case, as it explores all possible solutions.

– The brute-force algorithm has a time complexity of $O(2^{n*m})$, as it generates all possible configurations of traps and gems.

Conclusions:

- Scalability: The performance difference between algorithms becomes more pronounced as the problem size increases. While some algorithms perform very well for small problems (Brute-force), their performance deteriorates as the problem size grows.
- Performance Gap: There's a significant performance gap between the PySAT and CDCL algorithms compared to the backtracking and brute-force algorithms.
- Algorithm Choices: Depending on the problem size and time constraints, you would choose different algorithms:
  + For small problems:
    - PySAT, CDCL, and Backtracking are all viable options, with PySAT being the fastest.
    - Brute-force may also be considered due to its similar performance to Backtracking.
  + For larger problems:
    - PySAT remains the best choice, being significantly faster than other algorithms.
    - CDCL: Raw CDCL is still an acceptable method when comparing to PySAT as its upper bound is still exponential (as explained)
    - Backtracking and Brute-force are very slow and are not practical for large problems.
- In summary, PySAT is the most efficient algorithm across all problem sizes. Compare to PySAT, raw CDCL Algorithm implementation while still yeild a good enough result, is 7-8 times slower. This is due to the Tools in PySAT's library having other optimizations outside of just the main algorithm itself (Cache management, variable assignment, etc.), significant is the act of using **heuristic** for choosing variables and boolean assignments at the `PICK_VARIABLE` step. We did not use any heuristic to choose variable and corresponding value in the project as the performance, however 7-8 times slower comparing to PySAT, still less than 1 second in the game board size `66 x 66`. Backtracking and Brute-force algorithms are less efficient and become increasingly impractical as the problem size grows.

6 Self Evaluation

6.1 Project requirements

Requirements	**Completions**	**Notes**
Solution description	100%	None
Generate CNFs automatically	100%	None
Use pysat library to solve CNFs correctly	100%	None
Implement an optimal algorithm to solve CNFs without a library	100%	None
Program brute-force and backtracking algorithms	100%	None
Documents and other resources	100%	None
Write a report	100%	None

6.2 Member Evaluation

Student ID	Name	Tasks	Completions
22127357	Phạm Trần Yến Quyên	Use pysat library to solve CNFs Write the report	100%
22127402	Bế Lã Anh Thư	Program brute-force and backtracking algorithms	100%
22127459	Phạm Thanh Vinh	Write solutions descriptions Generate CNFs automatically	100%
22127488	Trương Thanh Toàn	Implement an algorithm to solve CNFs	100%

7 References

- CDCL algorithm:
  - CDCL SAT Solver from Scratch.
  - YouTube: CS443 SAT Solving.
  - Geeksforgeeks - CDCL.
  - University of Potsdam, Clause learning in SAT.
- PySAT - Official Documentations.
- Brute-force / Backtracking:
  - Lecture Notes on SAT Solvers & DPLL - Carnegie Mellon University
  - Exact Algorithms for General CNF SAT
  - Classes of cnf-formulas with backtracking trees of exponential or linear average order for exact-satisfiability