VIETNAM NATIONAL UNIVERSITY HCMC
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

———————o0o———————

Project 2: Gem Hunter

Subject: Introduction to Artificial Intelligence

Lecturer: **Dr. Nguyễn Tiến Huy**
Mr. Nguyễn Trần Duy Minh

Class: **22CLC05**

Group members: **22127010 – Đỗ Tân Ngọc Anh**
22127119 – Hồ Phước Hoàn
22127332 – Nguyễn Hoàng Phúc
22127394 – Lương Gia Thiếc

HCMC, 5/2024

Contents

Group Information

Group number: 15

No.	Student ID	Full name	Email
1	22127010	Đỗ Tân Ngọc Anh	dtnanh22@clc.fitus.edu.vn
2	22127119	Hồ Phước Hoàn	hphoan22@clc.fitus.edu.vn
3	22127332	Nguyễn Hoàng Phúc	nhphuc221@clc.fitus.edu.vn
4	22127394	Lương Gia Thiếc	lgthiec22@clc.fitus.edu.vn

Assignment Plan

No.	Task	In charge
1	Generate CNFs + Algorithm Implementation	Phước Hoàn
2	Use pysat library to solve CNFs correctly	Ngọc Anh
3	Implement brute-force algorithm	Gia Thiếc
4	Implement backtracking algorithm	Hoàng Phúc

Degree of Completion Level

No.	Requirements	Progress
1	**Solution description**: Describe the correct logical principles for generating CNFs.	100%
2	Generate CNFs automatically	100%
3	Use pysat library to solve CNFs correctly	100%
4	Implement an optimal algorithm to solve CNFs without a library	100%
5	Program brute-force algorithm to compare with your chosen algorithm (speed) Program backtracking algorithm to compare with your chosen algorithm (speed)	100%
6	**Documents and other resources that you need to write and analysis in your report**: Thoroughness in analysis and experimentation Give at least 5 test cases with different sizes (5x5, 9x9, 11x11, 15x15, 20x20. . .) to check your solution. Comparing results and performance	100%
Overall Completion		**100%**

1. Solution description

Describe the correct logical principles for generating CNFs.

Consider this configuration:

	0	1	2
0	**1**	_	_
1	_	_	**1**

To formulate CNF clauses, we first need to initiate variables. We will assign each empty cell '_' with a boolean variable $\boldsymbol{X_{i,\,j}}$ (***i***: row index, ***j***: column index), indicating if the cell **(*i*, *j*)** has a trap or not. In this state, we will be considering four unknown squares $\boldsymbol{X}_{0,\,1}$, $\boldsymbol{X}_{0,\,2}$, $\boldsymbol{X}_{1,\,0}$ and $\boldsymbol{X}_{1,\,1}$. Now comes the problem, how do we represent the global state CNFs for this board?

Consider the cell holds the value "**1**" at **(0, 0)**, we can immediately deduce that there will be exactly one trap in the surrounding cells $\boldsymbol{X}_{0,\,1}$, $\boldsymbol{X}_{1,\,0}$ and $\boldsymbol{X}_{1,\,1}$, or one of the three cells will be a trap. This can be represented in CNF as $(\boldsymbol{X}_{0,\,1} \vee \boldsymbol{X}_{1,\,0} \vee \boldsymbol{X}_{1,\,1})$. However, this clause only means "*at least one of the three cells is a trap*", in other words, there can be one, two or even three traps that can still satisfy the clause. We need more constraints added to ensure that there is "*at most one trap in the three cells*", since we are representing things logically in CNF, it could be quite complicated compared to DNF. The sentence "*at most one trap in the three cells"* is equivalent to "*at least two cells in the three cells are not traps*", now we can represent this sentence in CNF as:

$$(\neg\boldsymbol{X}_{0,\,1} \vee \neg\boldsymbol{X}_{1,\,0}) \wedge (\neg\boldsymbol{X}_{0,\,1} \vee \neg\boldsymbol{X}_{1,\,1}) \wedge (\neg\boldsymbol{X}_{1,\,0} \vee \neg\boldsymbol{X}_{1,\,1})$$

Combine the two constraints "*at least one of the three cells is a trap*" and "*at least two cells in the three cells are traps*", we can now formulate the complete CNF for the cell **(0, 0)** with value "**1**":

$$(\boldsymbol{X}_{0,\,1} \vee \boldsymbol{X}_{1,\,0} \vee \boldsymbol{X}_{1,\,1}) \wedge (\neg\boldsymbol{X}_{0,\,1} \vee \neg\boldsymbol{X}_{1,\,0}) \wedge (\neg\boldsymbol{X}_{0,\,1} \vee \neg\boldsymbol{X}_{1,\,1}) \wedge (\neg\boldsymbol{X}_{1,\,0} \vee \neg\boldsymbol{X}_{1,\,1})$$

Repeat the same procedure for every numbered tile, then take the conjunction of all generated clauses (in union, to eliminate all duplicate clauses), we will ultimately have the global CNF representation for the puzzle. Then this CNF can be used to solve the puzzle.

Above was only the resoning process to generate CNF for a single tile. Now we need to formulate a general formula for the algorithm to generate them automatically. Consider each numbered tile, we call $\boldsymbol{n}$ and $\boldsymbol{k}$ respectively as the unknown adjacent squares and unknown traps to be discovered in those $\boldsymbol{n}$ squares. Based on the logic that we build above, we call:

- $KN(n, k)$ as the CNF representation for a numbered tile.
- $M(k, n)$ means that at most $\boldsymbol{k}$ of the $\boldsymbol{n}$ squares contain traps.
- $L(k, n)$ means that at least $\boldsymbol{k}$ of the $\boldsymbol{n}$ squares contain traps.

As discussed, $M(k, n)$ is relevant to "*for any* $\boldsymbol{k+1}$ *cells in the* $\boldsymbol{n}$ *cells, at least one is* ***not*** *a trap*", consider any subset of $\boldsymbol{k+1}$ squares from $\boldsymbol{n}$ unknown squares, if at most $\boldsymbol{k}$ are traps, then at least one is not a trap. And $L(k, n)$ is relevant to "*for any* $\boldsymbol{n-k+1}$ *cells in the* $\boldsymbol{n}$*, at least one is a trap*", consider the rest of the subset of $\boldsymbol{n-k+1}$ squares, if at least $\boldsymbol{k}$ of $\boldsymbol{n}$ squares are traps, then the rest $\boldsymbol{n-k+1}$ squares will have at least one trap.

$$KN(n, k) \equiv (M(k, n) \wedge L(k, n))$$

Applying this formula for every numbered tile, then take them all into conjunction, we will have the same result, the CNF for the whole board. We can also recognize that these clauses can be generated by **permutations**, which made it much easier to implement in code.

For example, with this figure, our program can generate CNF as follows:

2. Algorithm analysis

2.1. Solving using PySAT

We are still using the formula mentioned in **Section 1**. However, for PySAT to be able to solve the puzzle, we need to follow its regulations. All logical variables in PySAT are represented integers, and for *NOT* variables (¬ negation), they are represented as negative integers.

For example, if we have a map like this:

```
-, -, -
1, 1, 1
```

CNFs for PySat will be generated as follow:

```
[[1, 2, 3], [-1, -2], [1, 2], [-1, -3], [-2, -3], [2, 3]]
```

equivalent to $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3)$

As can be seen, the number **1, 2** and **3** are integers representing the **empty cells** (unexplored cells) from left to right order, while the negative integers -1, -2 and -3 representing **negation (¬)** of that logical variable. There are 6 clauses in total, adhere to solution mentioned in **Section 1**. The CNFs clauses are stored in a list of lists, which is stored in a *cnf* variable. Then, we pass this variable to a SAT Solver in PySat (Glucose3) to solve. Finally, we will have our solution stored in the solver:

```
G, T, G
1, 1, 1
```

2.2. Brute-force algorithm.

This algorithm involves no CNFs presentation. The approach is simple, like its name, we generate all possible configurations of the puzzle, then check each of them to see if it is a valid configuration (similar to a goal-state test). However, this approach is not optimal since in the worst case senario, we might have to validate all $\mathbf{2^n}$ configurations (**n**: number of empty cells, each cell can hold either 'T' or 'G'), which is not suitable for solving this kind of puzzle in real time.

2.3. Backtracking algorithm.

Backtracking algorithm can be considered as an upgrade version from brute-force approach. We use the same idea – gradually assign values to empty cells then check if the configuration is valid. However, instead of filling out all cells at once, we fill the board out step by step, and at each step, we validate the current state of configuration. If the current assignment is not valid, then the algorithm traces back to the previous valid configuration, then begins from there with new assignments. With this new method, the algorithm can be surprisingly faster compared to brute-force for its ability to remove numerous invalid cases.

Although the speed has been significantly improved, this algorithm uses a lot of memory to save the configuration at each assignment.

2.4. CNFs solving algorithm.

This algorithm applies **Resolution** and **DPLL** algorithm approach.

Finding All Clauses:

- The find_all_clauses function generates all possible combinations of a given length(combi_len) from an input array (input_array).
- It uses a recursive approach to generate all combinations and stores them in the clause_tup list.
- This function is used to generate clauses representing the constraints in the Gem_hunter puzzle.

Finding Adjacent Cells:

- The find_adjacent_cells function takes the coordinates of a cell (i, j) and the dimensions of the board (row, col) as input.
- It returns a list of cell numbers representing the cells adjacent to the given cell.
- This function is used to identify the cells that should be included in the clauses based on the number constraints in the Gem_hunter puzzle.

Converting to CNF:

- The convert_to_CNF function takes the board, row, and col as input and generates the CNF clauses for the Gem_hunter puzzle.

- For each non-empty cell in the board, it generates two sets of clauses:
    - Clauses representing the constraint that at least n adjacent cells must be mines, where n is the number in the cell.
    - Clauses representing the constraint that at least k adjacent cells must not be mines, where k is the number of adjacent cells minus n.
- The function uses the find_adjacent_cells and find_all_clauses functions to generate these clauses.
- The resulting CNF clauses are returned as a list of lists.

Solving CNF using Resolution:

- The SolverCNF function takes a list of CNF clauses as input and applies resolution techniques to solve the problem.
- It first performs unit propagation to simplify the clauses based on any single-literal clauses.
- If a contradiction is found during unit propagation (i.e., a clause contains both a literal and its negation), the function returns an empty dictionary, indicating an unsatisfiable solution.
- If all clauses are satisfied after unit propagation, the function returns the current assignments.
- Otherwise, the function applies the DPLL algorithm to recursively assign truth values to literals and propagate the assignments to the clauses.
- The DPLL algorithm tries both assignments (true and false) for a literal and recursively solves the resulting CNF formulas.
- If a solution is found, the function returns the final assignments; otherwise, it returns an empty dictionary.

Applying Resolution:

- The apply_resolution function takes a list of CNF clauses as input and applies resolution techniques to solve the problem.
- It first performs unit propagation to simplify the clauses based on any single-literal clauses.

- If a contradiction is found during unit propagation, the function returns an empty dictionary, indicating an unsatisfiable solution.
- Otherwise, the function iteratively applies resolution steps to combine clauses and derive new clauses.
- If a clause with a single literal is derived, the corresponding assignment is made in the assignment dictionary.
- The process continues until no new clauses can be derived or all clauses are satisfied.
- The final assignments are returned as a dictionary.

Output Generation:

- The output function takes the board, solution (assignments), row, col, and a name as input.
- It generates a 2D list representing the solved Gem_hunter board.
- For each cell in the board, if the cell is unknown (-1), it checks the solution dictionary to determine whether the cell should be marked as a mine (T) or safe (G).
- The solved board is then written to a file named output{name}.txt.

3. Video demonstration

Here is the link to our demonstration video:

https://youtu.be/NPEoBq0DByg

4. Test results & Comparing performance

Here is the table contains information of the execution times we have recorded on **one** computer for each algorithm to solve puzzle with size of 5x5, 9x9, 11x11, 15x15 and 20x20, respectively (these maps are submitted together with the source code):

	CNF Solver	**Backtracking**	**Brute-force**
5x5	0.005015 seconds	0.002022 seconds	0.00956 seconds
9x9	0.030524 seconds	0.003524 seconds	606.470496 seconds
11x11	0.055347 seconds	0.005006 seconds	∞
15x15	0.218353 seconds	0.033573 seconds	∞
20x20	0.696429 seconds	0.052155 seconds	∞
20x20 (hard)	0.499085 seconds	15.608829 seconds	∞

Base on the collected data, we can draw to the conclusion that Brute-force is by far the worst algorithm, it will only performed well on a small scale of 5x5 and down, but above that will be costly along with downscale performance (notice that the infinity symbol represents that the algorithm takes too much time to solve the puzzle in real time, so we can ignore the result). Then came the fastest of the three - backtracking, it performed extraordinarily well, with the time solving increase very little, however, for the harder map of size 20x20 with more empty cells, it unfortunately took almost 16 seconds to solve, but still quite impressive. Then came CNF Solver using Resolution, it performed extremely well but a little less than Backtracking on the first five maps. However, the result for the last map was suprisingly weird, since this algorithm can solve this harder map faster than the previous 20x20 one, and far more optimal than Backtracking algorithm.

5. References

- Lecture 8 - people @ EECS at UC Berkeley. (n.d.).
  https://people.eecs.berkeley.edu/~daw/teaching/cs70-f03/Notes/lecture08.pdf

- Lecture 9 - people @ EECS at UC Berkeley. (n.d.).
  https://people.eecs.berkeley.edu/~daw/teaching/cs70-s05/Notes/lecture09.pdf

- Transforming DPLL to Resolution
  https://pure.tue.nl/ws/files/1766943/200207.pdf