VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

LAB REPORT

Gem Hunter

Presented by:	Lecturers
Trương Công Thiên Phú 23127455 23CLC07	Nguyen Ngoc Thao, PhD. Nguyen Thanh Tinh, MSc. Nguyen Tran Duy Minh, MSc. Ho Thi Thanh Tuyen, MSc

Ho Chi Minh City, April 7th 2024

Contents

1.Introduction:

1.1.Lab description:

The Gem Hunter is a game following these rules:

- Players explore a grid to find hidden gems while avoiding traps
- Each tile with a number represents the number of traps surrouding it. (Number from 1-8)

To find the solution for this game, the problem is formulated as CNF constraints and solved it using logic. By assigining logical value (True/False) for each empty tile, the clauses are generated through a function created by programmer and solved via pysat library. Brute-force and backtracking algorithms are implemented to solve these clauses and compare their speed (by measuring running time, which is how long it takes for a ocmputer to perform a specific task) and their performance with using the library.

Objectives:

- Formulate logical contraints.
- Generate CNFs automatically.
- Apply pysat library to solve CNFs correctly.
- Program brute-force algorithm to compare with using library.
- Program backtracking algorithm to compare with using library.
- Provide at least 3 test cases with different (5x5, 11x11, 20x20) to check the solution.
- Comparing results and performance.

1.2.File structure:

Here is the file structure in “Source code":

```
.
└── Source code/
    ├── Data/
    │   ├── DataHandler.py  # This file handle the data (including input data and output data)
    │   └── Display.py  # This file supports display the result to the screen
    ├── Tasks
    │   ├── Backtracking.py  # Backtracking algorithm
    │   ├── Bruteforce.py  # Brute-force algorithm
    │   ├── CNFs_Generation.py  # Generate CNFs automatically
    │   └── PySat.py  # pysat library solves CNFs
    ├── testcases  # This folder contains all solvable test cases there corresponding output/
```

```
│   ├── input_1.txt
│   ├── input_2.txt
│   ├── input_3.txt
│   ├── output_1.txt
│   ├── output_2.txt
│   └── output_3.txt
├── UI/
│   ├── CommandLine.py   # Handle command line arguments
│   └── Execution.py  # Execute the corresponding command from command line arguments
├── main.py
├── readme.txt  # Contain the introduction to run the program
└── requirement.txt  # Contain all the libraries to install
```

1.3.Evaluation of requirements completion level:

Requirements	Completion Rate
Decribe the correct logical principles for generating CNFs	100%
Generate CNFs automatically	100%
Use pysat library to solve CNFs correctly	100%
Program brute-force algorithm to compare with using library	100%
Program backtracking algorithm to compare with using library	100%
Documents and other resources are used to write and analysis in report	100%

1.4.Youtube Demo Video Link:

Link: https://youtu.be/xV8ewsc04-8

2.Locigal principles for generating CNFs:

A CNF (Conjuctive Normal Form) is a formula which its clauses are connected to each other by conjuction (logical AND), each caluse consists of literals expressed by dijunction (logical OR).

E.g: $(l1 \lor l2 \lor l3) \land (\neg l2 \lor \neg l5 \lor \neg l6) \land (l4 \lor l10 \lor l8) \lor \ldots$

We already know that surrounding a tile containing a number 'k', there are exactly k Traps. Base on this rules, the principles is determined by two keys:

- There are at most k-Traps for all empty cells surrouding the number cell.
- There are at least k-Traps for all empty cells surrouding the number cell.

Handle exactly k-Traps:

- At most k-Traps: To make sure no more than k 'cells' are true, the program generates all combinations of k + 1 literals. For each such combination, it adds a clause to the CNF that **negates all literals in the combination**. If k + 1 literals were true simultaneously, one of these clauses would be violated. Thus, this ensures that at most k literals can be true.
- At least k-Traps: To ensure that at least k 'cells' are true, the program generates all combinations of n + 1 - k literals (where n is the total number of atomic sentences). For each combination, it adds a clause containing the **original literals.** If all literals in any of these combinations were false, the clause would be unsatisfied. This guarantees that at least k literals must be true.

3.Programming Implementation:

3.1.CNFs generation:

Pseudocode

Function generate_CNF_s(grid) return a 2D-list
    #grid is a 2D-list
    Clauses = **<empty_list>**
    For each cell(i, j) in grid:
        If cell(i, j) is not '_' and cell(i, j) is not '0':
            Clauses += make_clauses(grid, i, j)
    #Avoiding duplicate
    Unique_clauses = **<empty_list>**
    For each clause in Clauses:
        If clause not in Unique_clauses:
            Unique_clauses.add(clause)
    Clauses = Unique_clauses
    Return Clauses

Function make_clauses(grid, i, j) return a 2D list
    Number_of_rows = length(grid)
    Number_of_cols = length(grid[0])
    Integer_value = int(grid[i][j]) *# Enforce from string to integer*
 	
    Atomic_sentence = **<empty_list>**
    For **each cell** surrounding grid[i][j]:
        If **this cell** is an <empty_cell>:
            Atomic_sentence.add(**this_cell.get_row()** * Number_of_cols + **this_cell.get_column()** + 1)
 	
    At_most = list(**<Integer_value + 1>** - combinations of Atomic_sentence)
    At_least = list(**<length(Atomic_sentence) +1 – Integer_value>** - combinations of Atomic_sentence)

```
        Return list(At_most + At_least)
```

Explain:

- Whenever a number cell is found, **make_clauses** function is invoked to generate all CNF clauses surrounding this cell.
- Follow this formula: surrounding_cell_value = <its_current_row> * <length of colums> + <its_current_column> + 1.
- Add these surrounding cell's value into a list called Atomic_sentence:
  - At_most is the <number_cell_value + 1> - combinations of Atomic_sentence.
  - At_least is the <length(Atomic_sentence) - number_cell_value + 1> - combinations of Atomic_sentence.

3.2.Pysat:

Before implementing this bellow function, remember to **import** Solver **from** pysat.solvers,

```
Pseudocode
Function pySat(grid, cnf_s):
        s ← Solver()

        for each clause in cnf_s:

           s.add_clause(clause)

        solvable ← s.solve()

        if solvable == True:

           model ← s.get_model()

           grid ← fill_result(grid, model)  # Function fill 'T' and 'G' symbols to the grid
        delete s

        return grid, solvable
```

Explain: This above pseudocode simulates how pysat library works

3.3.Brute-force algorithm:

```
Pseudocode
Function brute_force_SAT(cnfs, current_row, literals, solvable=True):
      if current_row == lenth(cnfs):  # Base case

           return <empty_list>, solvable


        previous_state ← solvable
```

 # Finding all literals which is not set to be True/False

undecided_literals ←{i for i in cnfs[current_row] if i not in literals.keys()}

undecided_literals ← list(undecided_literals)

list_unit_clauses ← **<empty_list>**

if **undecided_literals** is not **empty**:

for i in range(len(**undecided_literals**)):

list_unit_clauses += list(**<i + 1>** - combinations of **undecided_literals**)

for i in range(len(**list_unit_clauses**)):

list_unit_clauses[i] += [**-lit** from **lit** in **undecided_literals** if lit not in **list_unit_clauses[i]**]

list_unit_clauses += [[**-i** from **i** in **undecided_literals**]]

additional ← **<empty_dictionaries>**

position ← 0

if ***undecided_literals*** is not **empty**:
for **each literal** in **each clause** in **list_unit_clauses:**
literals[literal] ← **True**

literals[-literal] ← **False**

current_state ← **check_cnf**(cnfs[current_row], literals)

 # The above function (check_cnf) is invoked to check wheter a clause is True or False

next_state ← **[True]** if **previous_state** and **current_state** are **True** else [**False]**

additional, solvable ← **brute_force_SAT**(cnfs, current_row + 1, literals, next_state) ***# Recursive call to reach to the next cnf clause***

if **solvable** is **True**:

```
                position = i
                break
            else:
                for each literal in each clause in list_unit_clauses:
                    delete literals[literal]
                    delete literals[-literal]
    else:
        current_state ← check_cnf(cnfs[current_row], literals)


        next_state ← [True] if previous_state and current_state are True else [False]


        additional, solvable = brute_force_SAT(cnfs, current_row + 1, literals,
next_state)


    if solvable:
        result ← <emtpy_list>
        if undecided_literals:
            for lit in list_unit_clauses[position]:
                result.add(lit)


        result.extend(additional)
        return result, solvable
    else:
return <empty_list>, False
```

Explain:

- Parmeter <literals>: this is an dictionaries storing all literals value (E.g: literals[2] = True, literals[-2] = False)
- The idea to implement brute-force is that the program traverses all clauses and check whether a clause is **True** or **False**.

- For each traversal, the program gains all literals which is not set to logical value (**True/False)** and generate all possible combinations from 1 to n (where n is the number of literal which is not set to logical value in that clause).
- Recursive call is invoked to reach to the next clause whenever all literals in current clause is set to be **True** or **False** and that clause is checked its logical value.
- Despite of a **False** clause, recursive call is still invoked until it reachs to the base case.
- The **check_cnf** function traverses all literals in the clause and returns **True** if one of these literals is **True**, else it returns **False.**

3.4.Backtracking algorithm:

Pseudocode

```
Function back_tracking_SAT(cnfs, current_row, literals):
    if current_row == length(cnfs):
        return <empty_list>, True


    clause ← cnfs[current_row]
    satisfied ← False
    undecided_literals ← <empty_list>


    for each literal in clause:
        val ← literals.get(literal)
        if val is True:
            satisfied ← True
            break
        elif val is None:
            undecided_literals.append(literal)


    if satisfied is True:
        return back_tracking_SAT(cnfs, current_row + 1, literals)


    if undecided_literals is not empty:
```

```
    for each literal in undecided_literals:
        literals[literal] ← True
        literals[-literal] ← False

        res, solvable ← back_tracking_SAT(cnfs, current_row + 1, literals)
        if solvable is True:
            res += [literal]
            return res, True

        literals[literal] ← False
        literals[-literal] ← True

        res, solvable ← back_tracking_SAT(cnfs, current_row + 1, literals)
        if solvable is True:
            res += [-literal]
            return res, True

        delete literals[literal]
        delete literals[-literal]

    return <empty_list>, False
else:
    return <empty_list>, False
```

Explain:

- This algorithm is enhanced from Brute-force algorithm. Instead of setting all literals to logical value for each clause, this algorithm only set one literal is **True** and its negation is **False**. This is possible because a clause is **True** if one of its literals is **True**. Then recursive call is invoked and continues checking logical value for the next clause.

- If the recursive call returns **False**, the algorithm inverses the logical value of the literal which has been set
    - E.g: Before recursive call: literals[2] = True, literals[-2] = False
    - If recursive call returns **False**: literals[2] = False, literals[-2] = True
- …and try another approach to the next literal which is not set to logical value
- If the all approach in that clause is **False** the function immediately terminates that branch of approach

4.Test cases and output:

4.1.Size 5x5:

```
***pysat library solves CNFs***
Before solving:   After solving:
1, _, 3, _, 2  |  1, T, 3, T, 2
_, 4, _, 5, _  |  G, 4, T, 5, T
2, _, _, _, 3  |  2, T, T, T, 3
_, 4, _, 5, _  |  T, 4, T, 5, T
1, _, 2, _, 2  |  1, G, 2, T, 2

===>Executed Time: 0.00011(s)

***Backtracking solves CNFs***
Before solving:   After solving:
1, _, 3, _, 2  |  1, T, 3, T, 2
_, 4, _, 5, _  |  G, 4, T, 5, T
2, _, _, _, 3  |  2, T, T, T, 3
_, 4, _, 5, _  |  T, 4, T, 5, T
1, _, 2, _, 2  |  1, G, 2, T, 2

===>Executed Time: 0.00009(s)

***Brute-force solves CNFs***
Before solving:   After solving:
1, _, 3, _, 2  |  1, T, 3, T, 2
_, 4, _, 5, _  |  G, 4, T, 5, T
2, _, _, _, 3  |  2, T, T, T, 3
_, 4, _, 5, _  |  T, 4, T, 5, T
1, _, 2, _, 2  |  1, G, 2, T, 2

===>Executed Time: 0.00408(s)
```

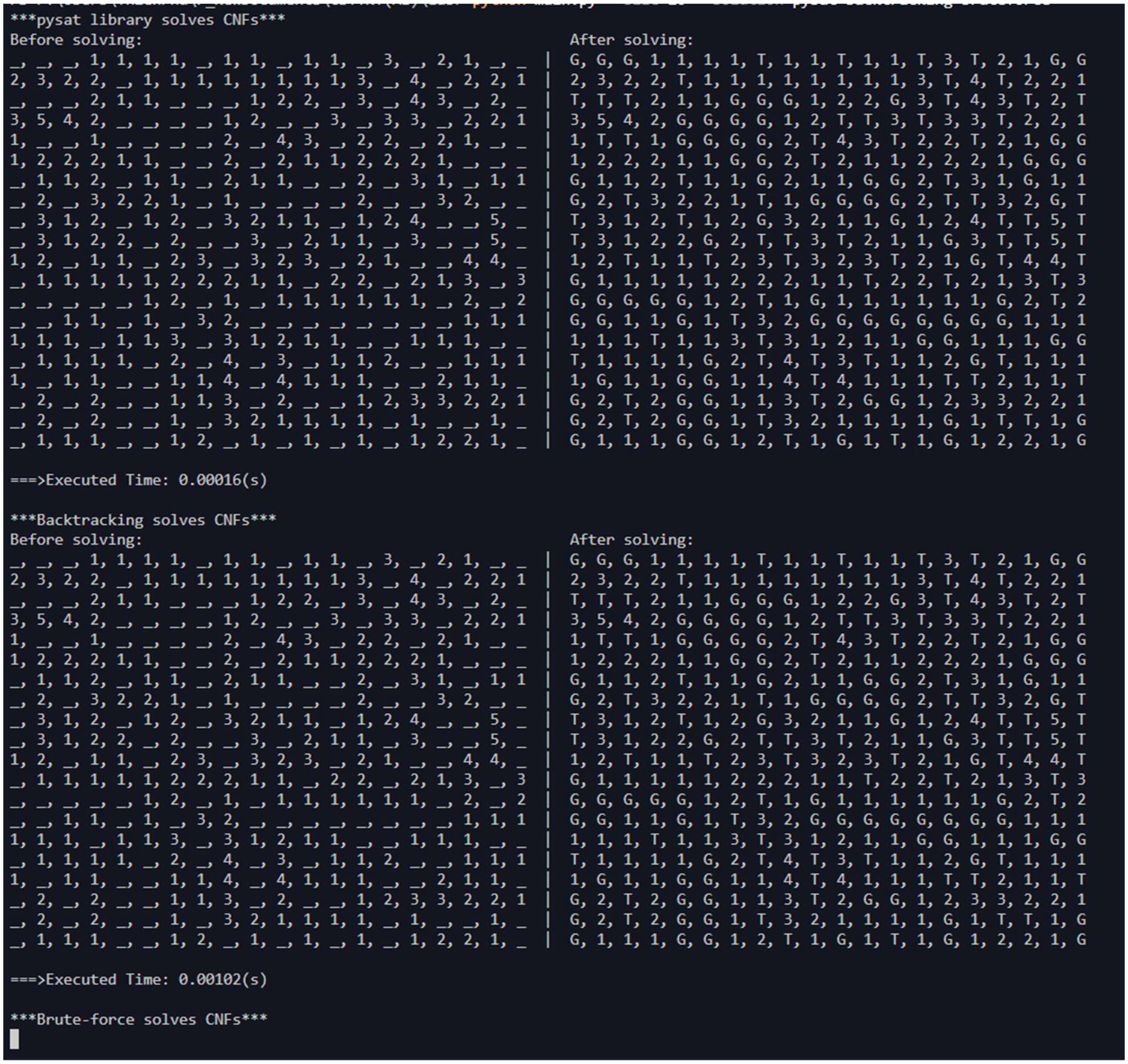
4.2.Size 11x11:

As Brute-force algorithm has to handle lots of cases (including the unsolvable cases), it would take a long time to find out the solution with this size.

```
***pysat library solves CNFs***
Before solving:                          After solving:
1, _, _, _, _, 1, 1, _, _, 1, _  |  1, G, T, G, G, 1, 1, G, T, 1, G
_, 2, 1, _, _, 1, _, 3, _, 2, _  |  T, 2, 1, G, G, 1, T, 3, G, 2, G
3, 4, _, 1, 1, 3, 3, 3, _, 1, _  |  3, 4, G, 1, 1, 3, 3, 3, T, 1, G
_, _, _, _, 1, _, _, 3, _, _, 1  |  T, T, T, G, 1, T, T, 3, G, G, 1
_, _, _, _, 2, 3, 2, 2, _, 2, _  |  T, T, G, G, 2, 3, 2, 2, T, 2, T
2, 3, _, 2, _, 2, 2, 3, 3, 3, 1  |  2, 3, G, 2, T, 2, 2, 3, 3, 3, 1
1, 2, _, _, 3, _, 2, _, _, 1, _  |  1, 2, T, G, 3, T, 2, T, T, 1, G
2, _, 3, _, _, 3, 3, 2, 2, 1, _  |  2, T, 3, T, G, 3, 3, 2, 2, 1, G
_, _, 4, 3, _, _, 2, 1, _, 1, 1  |  T, G, 4, 3, T, T, 2, 1, G, 1, 1
3, _, _, 3, 3, 4, _, _, 3, _, _  |  3, T, T, 3, 3, 4, T, G, 3, G, T
2, _, 3, _, _, 2, 2, _, _, _, 2  |  2, T, 3, G, T, 2, 2, T, T, T, 2

===>Executed Time: 0.00011(s)

***Backtracking solves CNFs***
Before solving:                          After solving:
1, _, _, _, _, 1, 1, _, _, 1, _  |  1, G, T, G, G, 1, 1, G, T, 1, T
_, 2, 1, _, _, 1, _, 3, _, 2, _  |  T, 2, 1, G, G, 1, T, 3, G, 2, G
3, 4, _, 1, 1, 3, 3, 3, _, 1, _  |  3, 4, T, 1, 1, 3, 3, 3, T, 1, G
_, _, _, _, 1, _, _, 3, _, _, 1  |  T, T, T, G, 1, T, T, 3, G, G, 1
_, _, _, _, 2, 3, 2, 2, _, 2, _  |  T, T, G, G, 2, 3, 2, 2, T, 2, T
2, 3, _, 2, _, 2, 2, 3, 3, 3, 1  |  2, 3, G, 2, T, 2, 2, 3, 3, 3, 1
1, 2, _, _, 3, _, 2, _, _, 1, _  |  1, 2, T, G, 3, T, 2, T, T, 1, G
2, _, 3, _, _, 3, 3, 2, 2, 1, _  |  2, T, 3, T, T, 3, 3, 2, 2, 1, G
_, _, 4, 3, _, _, 2, 1, _, 1, 1  |  T, T, 4, 3, T, T, 2, 1, G, 1, 1
3, _, _, 3, 3, 4, _, _, 3, _, _  |  3, T, T, 3, 3, 4, T, G, 3, G, T
2, _, 3, _, _, 2, 2, _, _, _, 2  |  2, T, 3, G, T, 2, 2, T, T, T, 2

===>Executed Time: 0.13044(s)

***Brute-force solves CNFs***
```

4.3.Size 20x20:

As Brute-force algorithm has to handle lots of cases (including the unsolvable cases), it would take a long time to find out the solution with this size.



5.References:

Generating CNF automatically: https://github.com/nguyentanhoangsa/CSAI-GemHunter/blob/master/source_code/createCNF.py

ChatGPT: https://chatgpt.com/

Pysat Documentation: https://pysathq.github.io/docs/html/

Video idea: https://youtu.be/WA_-mIzXGhw?si=Wg3Gp3pl6xNUr5zG