VNUHCM-UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

CSC14003 – INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Lab 02: Gem Hunter

Lecturer
Ms. Nguyễn Ngọc Thảo
Mr. Nguyễn Trần Duy Minh

Class
23CLC08

Students
23127226 – Nguyễn Đăng Minh

April 7th, 2025

Contents

1 Project Structure and Execution

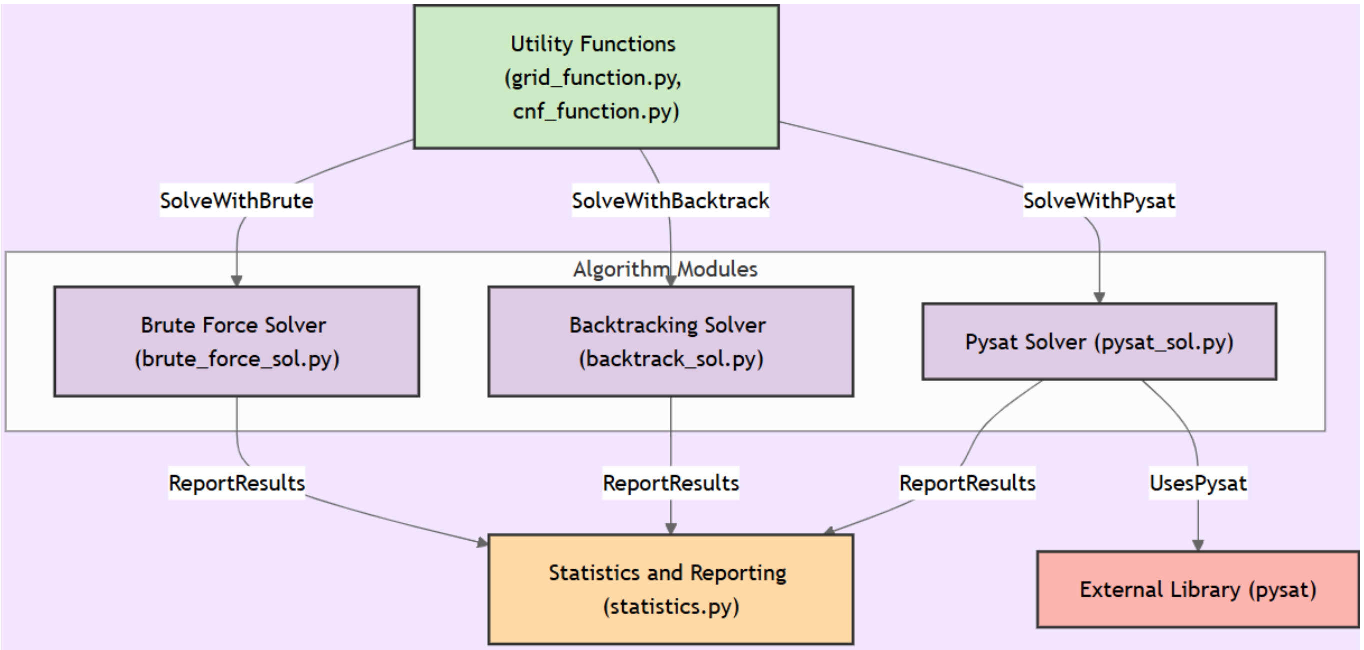

Figure 1: The code organization

The project organization is shown in Figure 1 above. There are three types of file category:

1. ***Util Files*** include functions of clauses, CNF, and grid that are used many times.
2. ***Algorithm Files*** will use different algorithms to solve the problem. All of them will include the ***Util Files*** to use the corresponding functions.
3. ***Statistics File*** gathers the result from algorithms and print the specific information of all algorithms along with the test cases. It is mainly used to experiment and compare between algorithms. Statistics are shown in section 6: Experiments.

2 Degree Of Completion Level

Assignment	**Subtask**	**Progress (%)**
CNF		
	Correct logical principles of CNFs	100%
	Generate CNFs automatically	100%
	Remove duplicate clauses	100%
PySAT		
	Use PySAT library to solve CNFs	100%
	Get the speed	100%
Backtrack		
	Use Backtrack with optimization to solve CNFs	100%
	Get the speed	100%
Brute Force		
	Use Brute Force to solve CNFs	100%
	Get the speed	100%
Algorithms comparison		
	Generate the statistics file	100%
	Comparison in the report	100%
Test cases		
	Have more than 3 test cases and they are varied in size.	100%
Video	Demo Running Process	100%
Report	Project Documentation	100%

3 Verifying The Results

In the process of coding, there must be a function to test if the result grid is valid in the Gem Hunter problem. The method of checking is simple as it will traverse through each number cell and count the number of traps around it. This ensures the confidence when comparing the algorithms as their results are believed to be accurate.
If it is false, then the result grid returned by some arbitrary algorithms is wrong. Below is the pseudo-code for validating the grid. The `trapCount()` function will calculate the number of traps in surrounding cells.

```
function CHECK_VALID_GRID(result_grid)
    for i = 0 to num_row do
        for j = 0 to num_col do
            if isNumberCell(i, j) and trapCount(i, j) ≠ result_grid[i][j] then
                return False
            end if
        end for
    end for
    return True
end function
```

4 Generating Conjunctive-Normal-Form Sentences

How the Conjunctive Normal Form sentences are created to solve the Gem Hunter problem?
First, I will denote that the value True of a variable means that there is a trap at the cell corresponding to that variable. Similarly, False value indicates the Gem.
In the Gem Hunter, I will use the number cell, number of traps surrounding it, as the **sensors** for the agent to solve the problem.

The easiest inference for the agent is that the number cell is neither the trap nor the gem cell. As the gem cells are not as important as the trap cell, the assignment value for the number cell will be False just like the gem. For example, in Figure 2 below, the $X2$ cell has value 2, so the CNF sentence added to the knowledge base is: $\underline{\neg X2}$.

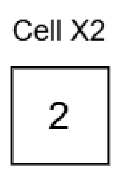

Figure 2: Number-cell case

How about the unknown cells? Let us consider a number cell with value x which indicates that there are exactly $\#x$ traps surrounding it in Figure 3 below.

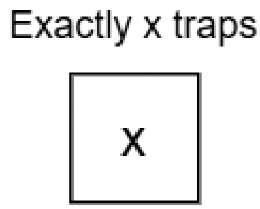

Figure 3: Exactly x traps in the cell

We can write sentences such as:
$\exists$ (combination of x cells out of 8) such that (x cells are traps) $\wedge$ ($8 - x$ cells are not traps)
This is true, however, it is very hard (or I have not found a way) to transform this sentence to Conjunctive Normal Form. Thus, I will divide the "Exactly x traps" sentence into two "weaker" sentences, "At least x traps" and "At most x traps", illustrated in Figure 4 below.

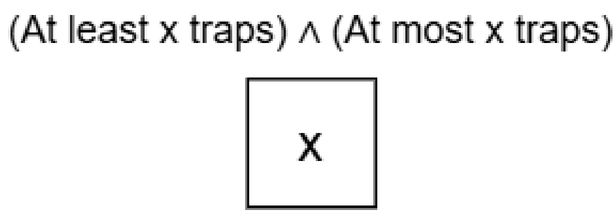

Figure 4: Two "weaker" sentences connected by **AND** logic

4.1 At-Most Sentence

Consider a situation in Figure 5 below.

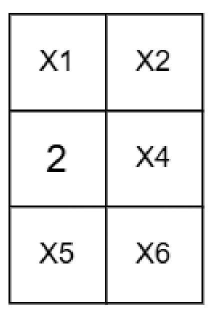

Figure 5: An example

The "At most 2 traps" sentence is actually the negation of "At least 3 traps". The "At least 3 traps" sentence corresponding to the above situation is expressed as below:
$(X1 \wedge X2 \wedge X4) \vee (X1 \wedge X2 \wedge X5) \vee (X1 \wedge X2 \wedge X6) \vee ... \vee (X4 \wedge X5 \wedge X6)$

It is basically all the combinations of 3 cells out of 6 neighbor cells connected via the $\vee$ symbol. `The bitmask-traversing method is used in this project to generate all valid possible combinations.` After negating the sentence, we get the final At-Most sentence of the CNF: $(\neg X1 \vee \neg X2 \vee \neg X4) \wedge (\neg X1 \vee \neg X2 \vee \neg X5) \wedge (\neg X1 \vee \neg X2 \vee \neg X6) \wedge ... \wedge (\neg X4 \vee \neg X5 \vee \neg X6)$

In conclusion, the "At most k traps" sentence can be transformed to CNF by negating the "At least $k+1$ traps" sentence.
"At most k traps" $\leftrightarrow \neg$"At least $k+1$ traps".

4.2 At-Least sentence

Consider the same example in Figure 5. If we perform the same operation as At-Most sentence: "At least k traps" $\leftrightarrow \neg$"At most $k-1$ traps", it would not yield the CNF because the At-Most sentence is already a CNF (proved above). The main problem is because of the **NOT** logic. Luckily, we can get rid of the NOT logic by slightly changing the definition:
"At least k traps" $\leftrightarrow$ "At most $n-k$ gems" (n is the number of **valid** neighbor cells)

Because gem is the "negation" of trap ($False = \neg True$), so we can take the same At-Most sentence and negate only the sign of the variables. For example, "At most 3 gems" will be expressed as:
$(X1 \vee X2 \vee X4) \wedge (X1 \vee X2 \vee X5) \wedge (X1 \vee X2 \vee X6) \wedge ... \wedge (X4 \vee X5 \vee X6)$

In conclusion, the "At least k traps" will be written as "At most $n-k$ gems" (n is the number of **valid** neighbor cells). Since "At most $n-k$ gems" $\leftrightarrow \neg$"At least $n-k+1$ gems", then "At least k traps" $\leftrightarrow \neg$"At least $n-k+1$ gems".

4.3 Factoring (Removing Duplicate Clauses)

After all the necessary clauses are added to the CNF sentence, the factoring procedure (removing duplicate clauses) are performed to optimize the clauses. Since the set of python will automatically remove duplicate elements, we first transform the clauses to set and create the official CNF sentence with after-factoring clauses. Below is the python code of the factoring process. Note that this is the idea of factoring, not the actual code in the project.

```python
    # Generate the CNF
    cnf = generated_cnf()
    # Convert to set
    factor_clauses = list(map(list, set(map(tuple, cnf.clauses))))
    # New factored CNF
    factor_cnf = CNF(from_clauses = factor_clauses)
    return factor_cnf
```

5 Gem-Hunter Solving Algorithms

5.1 Pysat Library Approach

Once we have generated the CNF sentence, we need to add all its clauses to the CNF() sentence of `PySAT` library [2]. Its format is: [[-1, 2], [3, -4]] indicates $(\neg X1 \vee X2) \wedge (X3 \vee \neg X4)$. The negative sign is similar to the $\neg$.
Initialize the `PySAT` solver with the CNF as the parameter. Then call the solve() function and get the model.
The **PySAT** solver will solve the CNF clauses incredibly fast. Even for more than 20000 clauses, it took only $0.0000ms$ to solve.
Below is the code to solve the CNF by **PySAT**:

```
solver = Solver(bootstrap_with = cnf)
if solver.solve():
    # Model format will contain all variables and their sign
    model = solver.get_model()
else:
    print("Unsatisfied")
```

5.2 Backtracking Approach

The backtracking algorithm will recursively try to assign value for each variables (only variables that stand for unknown cells). For each unknown variables which stand for unknown cells, first try to assign the True value, and if not yet succeed, backtrack again and attempt the False value. The test cases are made so that there always exists a solution.

However, a pure backtrack is similar to generating a whole truth table, which would yield the time complexity of $O(2^n)$. Thus, optimization is necessary. The optimization idea is alike to the DPLL algorithm [1], which applies early termination, pure symbol, and unit clauses heuristics. After applying the necessary heuristics, the backtracking algorithm works substantially fast as it could solve for a 30x30 grid (21094 clauses) in only 7 seconds.

Below is the pseudo-code for the backtracking algorithm with optimization:

Algorithm 1 Backtracking algorithm

```
function FIND_UNIT_CLAUSE(clauses, model) returns unit clause or null
    for clause in clauses do
        if No_True_Assignment(clause) and Unassigned_Var(clause) == 1 then
            return unassigned literal, sign
        end if
    end for
    return null
end function
----------------------------------------------------------------
function FIND_PURE_SYMBOL(clauses, model) returns pure literal or null
    for literal in clauses do
        if Only_One_Sign(literal) in clauses then
            return literal, sign
        end if
    end for
    return null
end function
----------------------------------------------------------------
function BACKTRACK(variables, clauses, model) returns True or False
    if exist false clause in clauses then return False          ▷ Early termination
    if all clauses in clauses true then
        save the model
        return True
    end if
    unitP, value ← Find_Unit_Clause(clauses, model)              ▷ Unit-clause heuristic
    if unitP ≠ null then
        return BACKTRACK(variables \ unitP, clauses, model ∪ {unitP = value})
    end if
    pureP, value ← Find_Pure_Symbol(clauses, model)              ▷ Pure-symbol heuristic
    if pureP ≠ null then
        return BACKTRACK(variables \ pureP, clauses, model ∪ {pureP = value})
    end if
    var ← FIRST(variables);
    return BACKTRACK(variables \ var, clauses, model ∪ {var = True}) or
           BACKTRACK(variables \ var, clauses, model ∪ {var = False})
end function
```

5.3 Brute Force Approach

Brute Force approach will generate all $O(2^n)$ possible bit-mask with n is the number of variables standing for unknown cells. Consider a simple situation below in Figure 6.

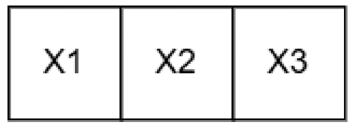

Figure 6: Brute Force example

Let us denote bit 1 in the bitmask be the trap, and 0 be the gem. For the example, we would have $2^3 = 8$ possible cases.
$[\{0,0,0\},\{0,0,1\},\{0,1,0\},\{0,1,1\},\{1,0,0\},\{1,0,1\},\{1,1,0\},\{1,1,1\}]$
For each mask that we generate, simply check if it satisfies the CNF. Here is the pseudo-code for the brute-force approach:

Algorithm 2 Brute Force

function `BRUTE_FORCE`($clauses, model$) **returns** mask or -1
  for $mask \in (2^n subsets)$ **do**
    if `SATISFIED`($clauses, mask$) **then**
      return mask
    end if
  end for
  return -1
end function

However, in the project, I performed the `"Iterative Brute Force"`, which iterates over the number of traps and check only masks that has the bit count of 1 equals number of traps. The pseudo-code for `"Iterative Brute Force"` is shown below. The `NEXT_MASK` function will get the new larger mask which have the same number of bit 1 count of current mask (if possible).

Algorithm 3 Iterative Brute Force (used in project)

```
function ITERATIVE_BRUTE_FORCE(clauses, model) returns mask or -1
    for trap_cnt = 0 → num_unknown_cells do
        mask ← (1 << trap_cnt)                      ▷ smallest mask of bit_count = trap_cnt
        while mask < (1 << num_unknown_cells) do
            if SATISFIED(clauses, mask) then
                return mask
            end if
            mask ← NEXT_MASK(mask)                  ▷ next larger mask with same bit_count
        end while
    end for
    return -1
end function
```

The overall time complexity of two algorithms are $O(2^n)$.

6 Experiments

6.1 Time Comparison

Instead of running each test cases, we can execute the `statistics.py` file and read all the statistics in the `statistics.txt` file. The statistics table is shown in Figure 7 after executing the file.

Grid size	# Clauses	# Empty cells	Algorithm	Time
5x5	496	11	pysat	0.0000ms
5x5	496	11	backtrack	1.2541ms
5x5	496	11	bruteforce	0.0000ms
6x6	621	24	pysat	0.0000ms
6x6	621	24	backtrack	2.5296ms
6x6	621	24	bruteforce	20.0171ms
10x10	1547	67	pysat	0.0000ms
10x10	1547	67	backtrack	48.8117ms
10x10	1547	67	bruteforce	Long
15x15	5695	129	pysat	0.0000ms
15x15	5695	129	backtrack	300.1463ms
15x15	5695	129	bruteforce	Long
20x20	10453	235	pysat	6.2177ms
20x20	10453	235	backtrack	1660.1064ms
20x20	10453	235	bruteforce	Long
30x30	21094	528	pysat	0.0000ms
30x30	21094	528	backtrack	6849.3955ms
30x30	21094	528	bruteforce	Long

Figure 7: Statistics Table

Based on the statistics table above:

- `PySAT`: in every cases PySAT is always the fastest one. The overall time in milliseconds for the Gem-Hunter solver using PySAT is 0.0000ms, even for the 30x30 grid (21094 clauses), which is incredible.
- `Backtrack`: The Backtracking algorithm with optimization also performs well as the search space is effectively pruned. For the largest test case, backtrack only took nearly 7 seconds to solve.
- `Iterative Brute Force`: Iterative Brute Force approach is the longest one and there is no optimization applied. As mentioned in section 4.3, the time complexity for Brute-Force algorithm is $O(2^n)$. Moreover, I have tested for many other test cases and found out that Iterative Brute Force seems to be slightly faster than Brute Force approach. In conclusion, Brute Force algorithms can only run in reasonable time if the number of unknown cells are less than 25.

6.2 Result Verifying

Let us consider the result for the 20x20 test cases (test case 5).

```
1 _ _ _ 2 2 2 1 1 1 2 1 1 1 _ 2 1 _ 2 2
_ _ _ 4 3 _ _ 2 2 _ _ _ 1 _ _ _ _ 3 _ _
_ 4 2 _ _ _ _ 3 _ 3 _ _ _ 3 _ 3 _ 4 _ 4
_ 2 _ _ _ 2 _ _ _ 2 _ 3 _ _ 4 _ 1 4 _ 3
2 3 3 _ _ 2 _ 1 _ _ _ _ 5 _ _ 2 1 4 _ 4
_ _ _ 2 _ 1 2 2 2 3 _ _ _ _ _ _ _ 4 _ _
3 4 3 _ _ _ _ _ _ 3 _ 3 1 1 1 _ 2 _ _ _
_ _ 1 _ _ 3 3 2 2 2 _ _ 1 1 _ _ 1 1 2 2
_ 3 2 _ _ _ 3 2 _ 1 _ 2 _ 3 2 3 2 1 _ _
_ _ _ 2 _ _ 4 _ _ 2 _ 3 _ 4 _ 2 _ _ 2 _
_ _ _ 5 _ _ _ _ 4 _ _ _ _ 4 _ _ _ _ _ _
2 _ _ _ _ _ _ _ _ _ _ _ 3 _ _ _ 2 1 _ 1
1 2 4 _ _ 3 4 _ 3 2 4 _ _ 4 _ _ _ _ _ _
2 _ _ 1 3 _ _ _ _ _ _ _ _ _ _ 3 _ 2 _ _
_ 2 1 _ 2 _ _ 4 4 5 _ _ _ _ 4 _ 3 _ 2 1
1 1 _ 1 3 4 _ 2 _ _ 5 _ 3 2 _ _ _ 3 _ _
_ _ _ _ _ _ 3 4 _ 6 _ _ 2 _ 2 _ _ _ 3 _
_ _ _ 1 2 2 3 _ _ _ _ _ 2 1 1 _ _ 4 _ _
_ _ _ _ 2 1 2 _ _ 5 4 4 _ _ 2 _ _ 4 3 _
_ _ 1 _ _ _ 1 2 _ _ _ 2 _ 3 _ 3 _ _ _ 2
```

Figure 8: Test case 5

Below is the result grid by PySAT and Backtrack algorithm:

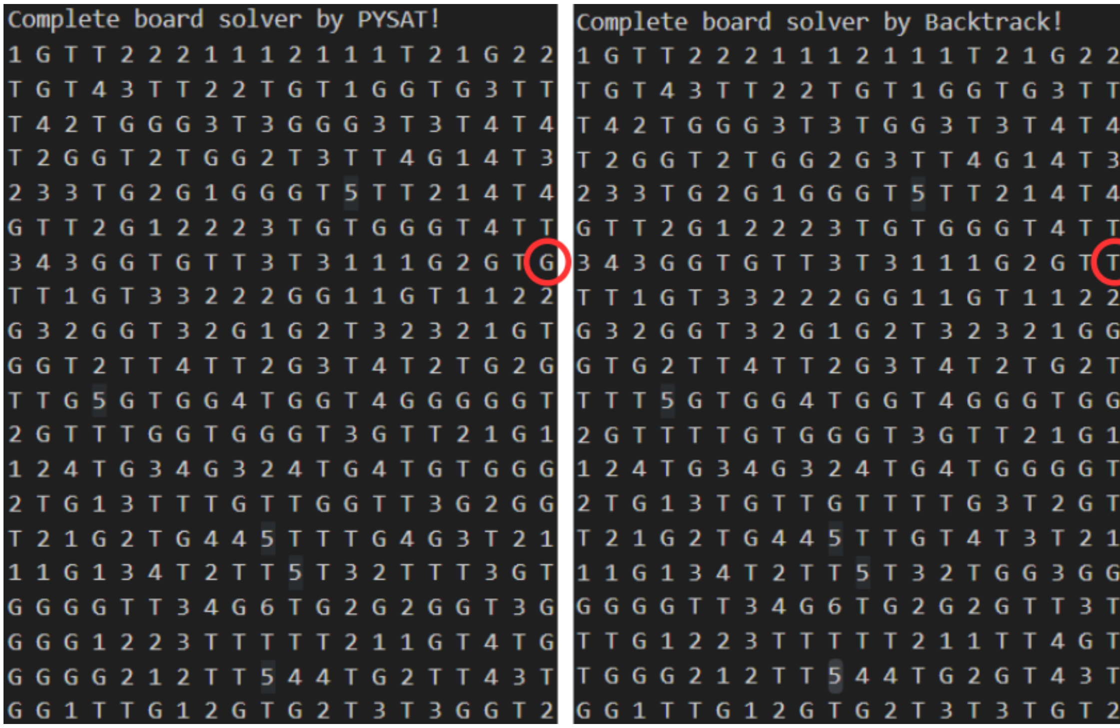

Figure 9: Result of PySAT and Backtrack

Since the result grid generated by Backtrack algorithm has been verified by the `CHECK_VALID_GRID` function in section 2, it is ensured to be accurate. Besides, if we look closely, the result of backtrack is slightly different from PySAT, which is marked by the red circle.

References

[1] Wikipedia. *DPLL algorithm*. URL: https://en.wikipedia.org/wiki/DPLL_algorithm.

[2] Wikipedia. *PySAT Documentation*. URL: https://pysathq.github.io/docs/html/api/solvers.html.