HO CHI MINH UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

REPORT: Project 02: Gem Hunter

By:
22127057 – Đỗ Phan Tuấn Đạt
22127064 – Phạm Thành Đạt
22127123 – Lê Hồ Phi Hoàng
22127131 – Trần Nguyễn Minh Hoàng

A.TABLE OF CONTENTS

B.INTRODUCTION

This project was initialized as part of the course CSC14003 – Introduction to Artificial Intelligence. Its purpose is to help the students learn the basics of creating and implementing artificially intelligent agents through the simple puzzle game Gem Hunter, which has multiple similarity with Minesweeper.

CNF (Conjunctive Normal Form) is utilized in all strategies implemented in the program, including Brute Force, DPLL, WalkSAT, and solving with Python-sat module. This offers unique insights into the efficiency and effectiveness of different approaches to problem-solving within the context of Gem Hunter.

Through this project, students not only gain hands-on experience in developing AI agents but also cultivate critical thinking skills essential for addressing complex real-world problems. We believe that the insights gained from this endeavor will serve as a solid foundation for future endeavors in the field of artificial intelligence.

C.WORKING ENVIRONMENT

I. Programing language

- 100% Python.
- External modules were used. Please have the below modules installed before running the program:
    - python-sat

II. Code editor

Visual Studio Code

III. Code management

GitHub

D. USER INSTRUCTIONS

- cd to the directory of the program.
- Simply run the main.py file.
- Choose an algorithm and an input file by inputting their corresponding numbers.
- Wait up to 5 minutes for the calculations. If the program does not continue, that means it will take a lot more time to calculate. At that point, you can decide whether to terminate the program.
- In case you want to run with your own input file, put the file inside “testcases” folder (.txt files only) before running the program. You should be able to select it as the input.
- For more detailed instructions, watch here: https://www.youtube.com/watch?v=kixwqOj62VY

E. GENERATING CNF

I. Idea

- Each cell is represented by a unique variables via dictionary.
- Representing AND/OR operations using lists and nested lists example:
  - [[1], [2]] = 1 ∧ 2
  - [1, 2] = 1 ∨ 2
- Numbered cells: cells that hold values indicating the number of traps located adjacent to a specific cell within a 1-block range.
- Surrounding cells: Empty cells that are yet to be determined whether they are Gems/Traps in relation to a numbered cell.
- Gem are understood as True, and Traps are understood as False in the context of clause creation.
- Main idea of automatically generating CNFs:

2	1
2	3

- In this example: The numbered cell has a value of 2 indicating that there are 2 cells from 1, 2, 3 that are traps.
- This means that there are 3C2 = 3 combinations that needs to be considered.
  - If 1 and 2 are traps which imply that 3 is holds gem and vice versa: (¬1 ∧ ¬2) ⇔ 3 (The same clause can be generated for the other two instances).
    - This can be converted to CNF as follows:
      (1 ∨ 2 ∨ 3) ∧ (¬1 ∨ ¬2 ∨ ¬3)
    - Then we can convert this into the appropriate format:
      [[1, 2, 3], [-1, -2, -3]]

3	1
2	3

- In special cases when the value of the numbered cell is equal to the number of empty cells surrounding it, we can immediately derive the following clause: (¬1 ∧ ¬2 ∧ ¬3)
- Converting to CNF in the correct format:
  [[-1], [-2], [-3]]

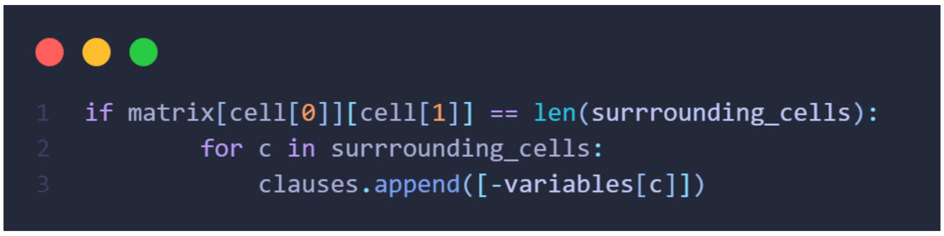
- Duplicates removal: The clause generation method mentioned above will obviously cause a lot of duplicates to be formed, and the best way to resolve this issue is to sort the list and then check for duplicates before accepting the clause.
  - For example: if [[1, 2], [3, 4]] is the current clause, and a [2, 1] clause has just been generated, a conversion is made from [2, 1] to [1, 2], which already exists inside the original clause. Therefore, the new clause will not be added.

  - After the clause generation process is complete, we can loop over the entire clauses one more time to remove duplicates that still remain.
- <u>**Note:**</u> However, this will pose a slight issue when it comes to cells that do not have any numbered cells adjacent to them. This means that no restrictions are imposed on these cells, and thus, assigning Gem as a default value to these 'irrelevant' cells is valid.

II. Implementation

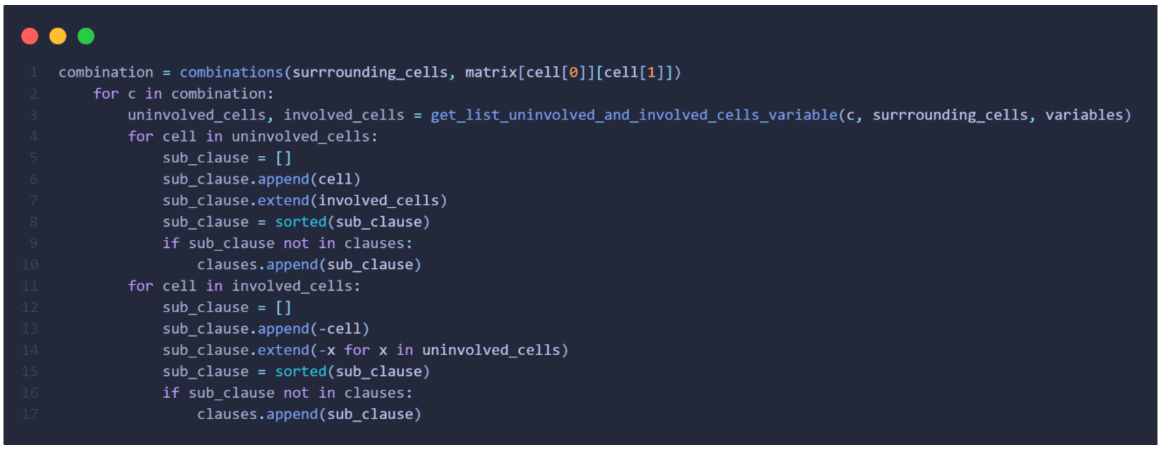
- **assign_variables(matrix, num_rows, num_cols)**: Assigns variables to each cell in the matrix
  - Generates a unique variable identifier for each cell in the matrix.
  - Initializes a dictionary mapping each cell coordinate to its corresponding variable identifier.
  - Initializes a dictionary to store the truth values of variables.
  - Returns dictionaries mapping cell coordinates to variable identifiers and variable identifiers to truth values.
- **get_surrounding_cells(matrix, pos, num_rows, num_cols)**: Retrieves neighboring cells of a given cell
  - Takes a cell position **pos** and the dimensions of the matrix.
  - Determines the neighboring cells (up, down, left, right, and diagonals) of the given cell.
  - Returns a list of neighboring cell coordinates.
- **get_numbered_cells(matrix, num_rows, num_cols)**: Identifies cells with numbered values by iterating through the matrix and identifies cells with non-None values. Then returns a list of coordinates of cells with numbered values.
- **get_irrelevant_cells(matrix, num_rows, num_cols)**: Return a list of cells that have no constraints imposed upon them (Cells that are special cases which do not appear during clause generation as noted above).
- **get_list_uninvolved_and_involved_cells_variable(combination, surrounding_cells, variables)**: Separates involved and uninvolved cells in a combination
  - Takes a combination of cells and surrounding cells.
  - Determines which cells in the surrounding area are involved in the combination and which are not.
  - Returns lists of variables representing uninvolved and involved cells.

- **generateCNFFromConstraintsByCell(cell, matrix, num_rows, num_cols, variables)**: Generates CNF clauses based on constraints around a cell
    - Get all of the surrounding cells via **get_surrounding_cells** function.
    - If value of numbered cells equal to the number of surround cells than we can safely assume that all of these cells are traps.



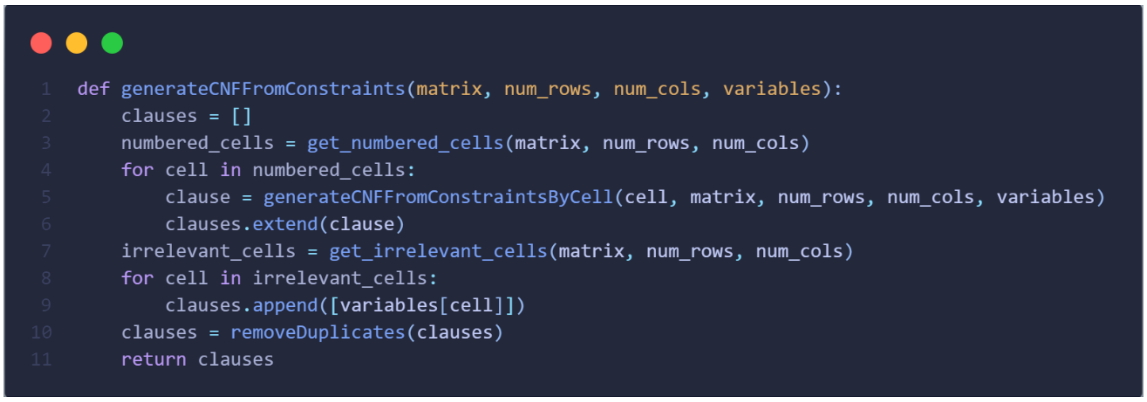
```python
if matrix[cell[0]][cell[1]] == len(surrounding_cells):
        for c in surrounding_cells:
            clauses.append([-variables[c]])
```

    - If not, then we get all possible combinations of cells that could possibly be traps, and iterate over each combination to form CNF clauses as mentioned in the section above while also accounting for duplicates.



```python
combination = combinations(surrounding_cells, matrix[cell[0]][cell[1]])
    for c in combination:
        uninvolved_cells, involved_cells = get_list_uninvolved_and_involved_cells_variable(c, surrounding_cells, variables)
        for cell in uninvolved_cells:
            sub_clause = []
            sub_clause.append(cell)
            sub_clause.extend(involved_cells)
            sub_clause = sorted(sub_clause)
            if sub_clause not in clauses:
                clauses.append(sub_clause)
        for cell in involved_cells:
            sub_clause = []
            sub_clause.append(-cell)
            sub_clause.extend(-x for x in uninvolved_cells)
            sub_clause = sorted(sub_clause)
            if sub_clause not in clauses:
                clauses.append(sub_clause)
```

    - Returns CNF clauses and a flag indicating if the cell is fully satisfied.
- **generateCNFFromConstraints(matrix, num_rows, num_cols, variables, unit_clauses=[])**: Iterate over all of the numbered cell and call generateCNFFromConstraintsByCell to get clauses. In addition to that, append cells that is defined as 'irrelevant' in the above section as Gems.

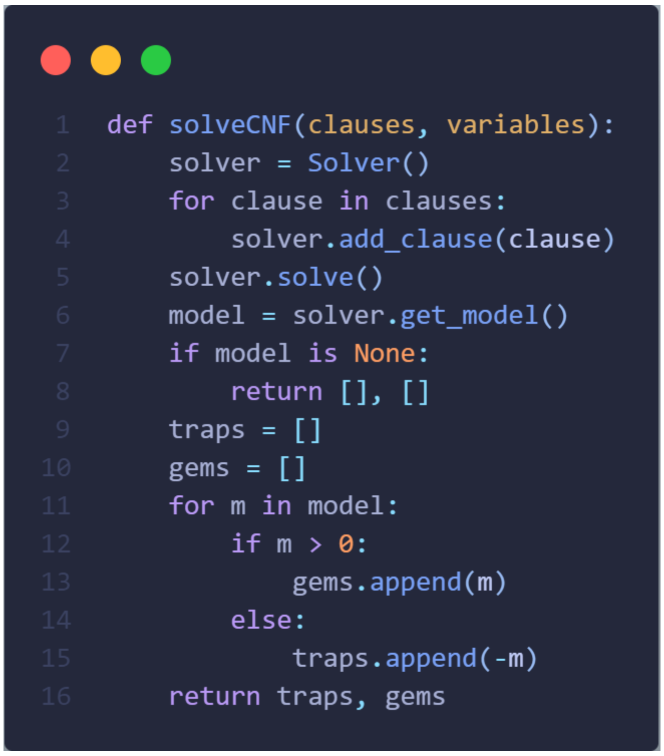


```python
def generateCNFFromConstraints(matrix, num_rows, num_cols, variables):
    clauses = []
    numbered_cells = get_numbered_cells(matrix, num_rows, num_cols)
    for cell in numbered_cells:
        clause = generateCNFFromConstraintsByCell(cell, matrix, num_rows, num_cols, variables)
        clauses.extend(clause)
    irrelevant_cells = get_irrelevant_cells(matrix, num_rows, num_cols)
    for cell in irrelevant_cells:
        clauses.append([variables[cell]])
    clauses = removeDuplicates(clauses)
    return clauses
```

- **removeDuplicates(clauses)**: Removes duplicate clauses from a list

F. PUZZLE-SOLVING ALGORITHMS

I. Solve using python-sat module



```python
def solveCNF(clauses, variables):
    solver = Solver()
    for clause in clauses:
        solver.add_clause(clause)
    solver.solve()
    model = solver.get_model()
    if model is None:
        return [], []
    traps = []
    gems = []
    for m in model:
        if m > 0:
            gems.append(m)
        else:
            traps.append(-m)
    return traps, gems
```

- Default solver if not declared is ‘m22’
- Model from get_model return a list of all variable in its True form:
  - If 1 is true then 1 is in model
  - If 1 is false then -1 is in model
- Then, we can separate variables that are considered to be traps (False or < 0), and Gems (True or > 0)
- solutionMatrix: from the original matrix, and the lists of traps/gems, create a new matrix in which all of the empty cells are replaced by an appropriate symbols representing trap/gem, while the rest remain unchanged.

II. Brute-force Solving (Backtracking)

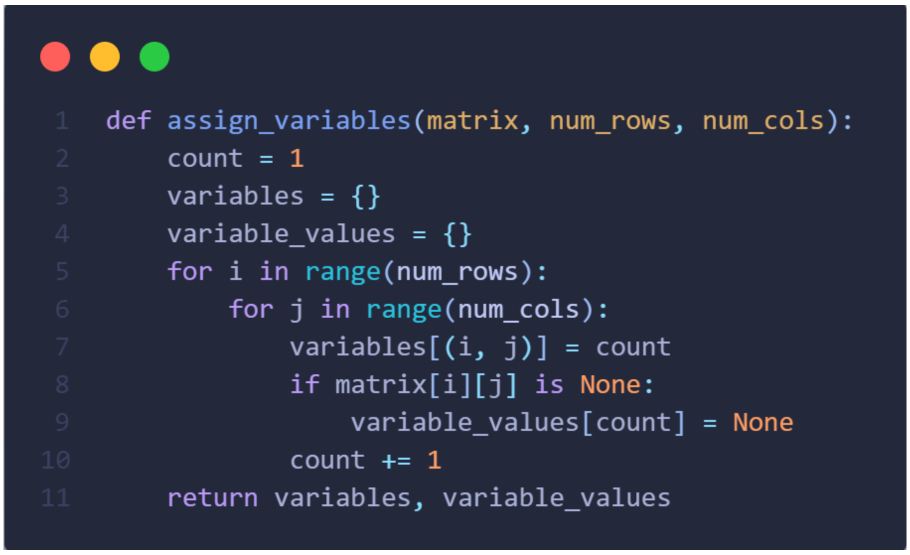
1. Idea:

- If the value of the numbered cell is equal to the number of surrounding cells, then all surrounding cells are automatically **False** (or is considered to be a trap).

- If the empty cell is not affected by any numbered cells, then that cell is automatically set to **True** (or is considered to be a gem).
- If not solve, run all permutation of all the remaining variables that have yet to be assigned.
- After each permutation, check to see if all the variables fit all of the constraints presented by the numbered cells, if yes then there exist a solution, otherwise, backtrack to previous state and continue running. If all of the permutations have been check and none satisfy the constraints, return **False**.

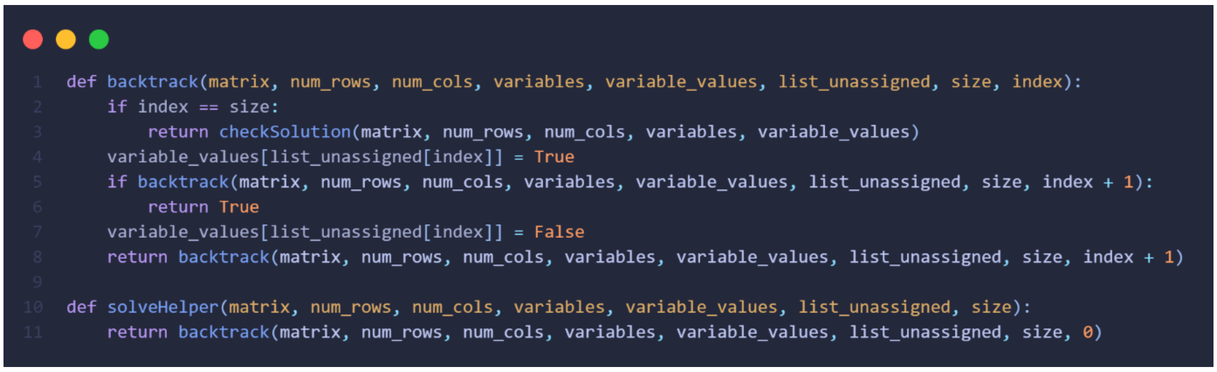
2. Implementation:

- Minor adjustment: **assign_variables**: Add a *variable_values* which will be initialized to **None** for all variables. This, will later be used to set variable to **True**/**False** without affecting the actual variables.



```python
def assign_variables(matrix, num_rows, num_cols):
    count = 1
    variables = {}
    variable_values = {}
    for i in range(num_rows):
        for j in range(num_cols):
            variables[(i, j)] = count
            if matrix[i][j] is None:
                variable_values[count] = None
            count += 1
    return variables, variable_values
```

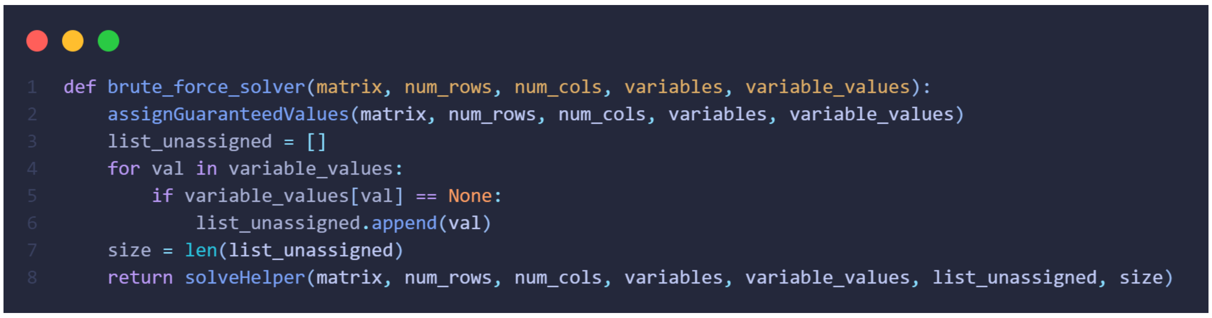
- **assignGuaranteedValues**: Process special cases (value of numbered cell = number of neighboring cell; Empty cells that have no constraints)
- **checkSolution**: Check to see if value of variables (**True**/**False**) satisfy all of the numbered cells requirements.
- **backtrack**: This is a recursive function used to try all possibilities of unassigned variables to find a solution.
    - It tries assigning **True** to the next variable in the list of unassigned variables. If a solution is found, it returns **True**.
    - If not, it tries assigning **False** to the next variable and continues recursively. If a solution is found, it returns **True**.
    - If neither case is successful, it means there is no solution, and the function returns **False**.



```python
def backtrack(matrix, num_rows, num_cols, variables, variable_values, list_unassigned, size, index):
    if index == size:
        return checkSolution(matrix, num_rows, num_cols, variables, variable_values)
    variable_values[list_unassigned[index]] = True
    if backtrack(matrix, num_rows, num_cols, variables, variable_values, list_unassigned, size, index + 1):
        return True
    variable_values[list_unassigned[index]] = False
    return backtrack(matrix, num_rows, num_cols, variables, variable_values, list_unassigned, size, index + 1)

def solveHelper(matrix, num_rows, num_cols, variables, variable_values, list_unassigned, size):
    return backtrack(matrix, num_rows, num_cols, variables, variable_values, list_unassigned, size, 0)
```

- **brute_force_solver**: Combining all of the above functions to solve the matrix.



```python
def brute_force_solver(matrix, num_rows, num_cols, variables, variable_values):
    assignGuaranteedValues(matrix, num_rows, num_cols, variables, variable_values)
    list_unassigned = []
    for val in variable_values:
        if variable_values[val] == None:
            list_unassigned.append(val)
    size = len(list_unassigned)
    return solveHelper(matrix, num_rows, num_cols, variables, variable_values, list_unassigned, size)
```

III. DPLL

1. Idea

- At the core of the DPLL algorithm is still backtracking, but there are improvements which can be made to increase the speed of the algorithm.
- Important improvements:
- At each iteration, a literal is made **True**, this will alters the clauses created previously during the CNF generation phase.
    - If literal L1 is true, the clause (L1 ∨ L2 ∨ …) is true. Therefore, there is no need to go through the rest of this clause, and can be removed from the list of clauses.
    - If clause C1 is true, then C1 ∧ C2 ∧ C3 ∧… has the same value as C2 ∧ C3… Therefore, removing C1 does not affect the final result of the clause and should be shorten.
- If the derived list of clauses after this process contains an empty clause (or in the correct format: [[]]) then the assignment is incorrect, resulting in backtracking to the previous iteration to assign the literal to False.
- If there exist no clause left after this process (or in the correct format: []) then a solution has been found.
- Minor Improvements:
    - In order to shorten running times, unit literals can be prioritized.
    - Unit literal is a literal that appears in a singleton clause.
    - If C1 is false, then C1 ∧ C2 ∧ C3 ∧… will also be false. This means

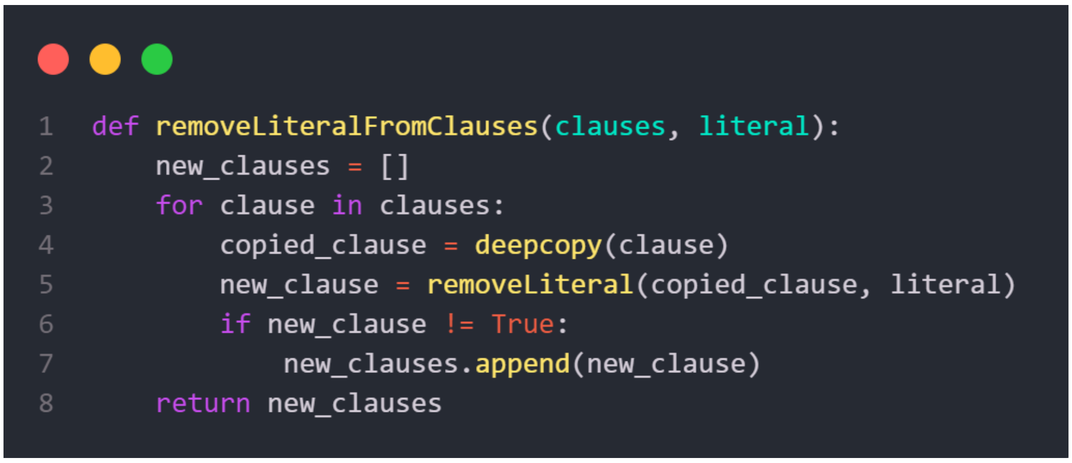
that unit literals must automatically be set to true.

- For example: [[-b, c], [-c], [a, -b, e], [d, b], [e, a, -c]]
    - [-c] is a unit clause so it is to **True** (meaning c is **False**)
    - The clauses after processing: [[-b], [a, -b, e], [d, b]]
        - [-b] is a unit clause so it is set to **True** (meaning b is **False**)
        - The clause after processing: [[d]]
            - [d] is a unit clause so it is set to **True** (meaning d is **True**)
    - In this example, we can see that when we shorten clauses and implements unit clauses, there exists multiple cases in which backtracking is never initiated, making it significantly faster than normal backtracking search.
- MOM's heuristic:
    - Definition: Most Occurring Literal in Minimum Length Clauses.
    - This heuristic is to prioritize finding the most commonly occurring literal within the shortest clauses because in doing so, we aim to create as many unit clauses as possible in each iteration of the algorithm, resulting is faster reduction of the overall set of clauses (if unit clause exists in clauses then it will be found using this heuristic as well).
- **<u>Note</u>**: In the process of generating Conjunctive Normal Form (CNF), each variable will have two versions: one that's positive and one that's negative. For example, if we have a variable x, its positive literal would be x and its negative literal would be ¬x. Since both versions are presented, there's no need to worry about pure literals, which are variables that only appear positively or negatively but not both.

2. Implementation

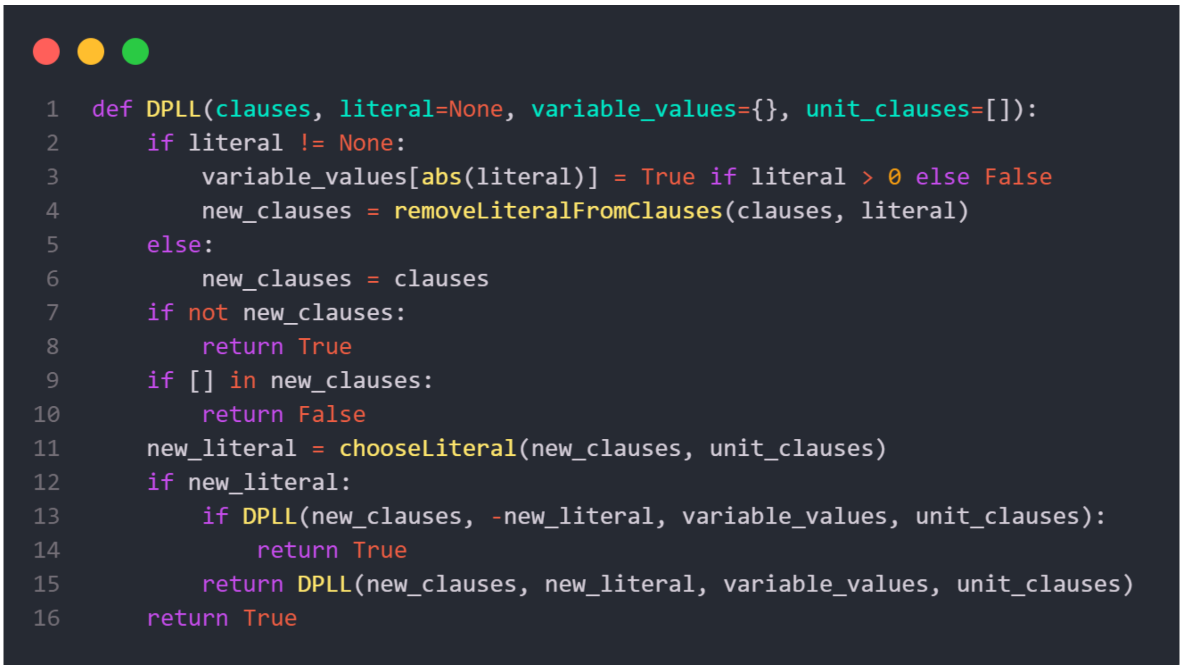
- Minor adjustments:
    - **assign_variables**: The same as bruteforce.
    - **generateCNFFromConstraintsByCell**: There is an additional return value added: a boolean indicating whether the clauses generated are unit clause.
    - **generateCNFFromConstraints**: An additional parameter unit_clauses is added to the function signature to keep track of unit clauses separately from the main clauses list.
- **removeLiteral**: This function takes a clause (a list of literals) and a literal as input. It checks if the literal or its negation is present in the clause. If the literal is found, it returns **True**, indicating that the clause is satisfied. If the negation of the literal is found, it removes it from the clause. Finally, it returns the modified clause.

- **removeLiteralFromClauses**: This function removes a literal from all clauses in a list of clauses. It iterates over each clause, creates a deep copy to avoid modifying the original, and calls **removeLiteral** to process each clause. If **removeLiteral** returns **True**, indicating that the clause is satisfied after removing the literal, the clause is not added to the new set of clauses. Otherwise, the modified clause (without the removed literal) is added to the new set of clauses. The function returns the new set of clauses.



```python
def removeLiteralFromClauses(clauses, literal):
    new_clauses = []
    for clause in clauses:
        copied_clause = deepcopy(clause)
        new_clause = removeLiteral(copied_clause, literal)
        if new_clause != True:
            new_clauses.append(new_clause)
    return new_clauses
```

- **calcOccuringLiteralInMinClauses**: This function calculates how many times a specific literal (value) occurs in a set of clauses (*min_clauses*). It initializes a counter and iterates over each clause in *min_clauses*. If the literal is found in a clause, the counter is incremented. Finally, it returns the count of occurrences of the literal.
- **getMostOccuringLiteral**: This function finds the literal that occurs most frequently in a set of clauses (*min_clauses*). It initializes variables to keep track of the maximum count (*count*) and the corresponding literal (*res*). It iterates over each literal in the set val. For each literal, it calculates the number of occurrences in *min_clauses* using **calcOccuringLiteralInMinClauses**. If the count is greater than the current maximum count, it updates *count* and *res* accordingly. Finally, it returns the literal that occurs most frequently.
- **chooseLiteral**: This function selects a literal to branch on during the DPLL algorithm. If there are unit clauses available, it returns the first literal in the first unit clause and removes that clause from *unit_clauses*. Otherwise, it identifies the minimum clause length (*min_len*) among all clauses. It then identifies the literals occurring in clauses of length *min_len* and selects the one that occurs most frequently across those clauses using **getMostOccuringLiteral**. If no literals are available (e.g., all clauses are satisfied), it returns **None**.
- **dpll**: Combining all of the above functions to solve the matrix.



```python
def DPLL(clauses, literal=None, variable_values={}, unit_clauses=[]):
    if literal != None:
        variable_values[abs(literal)] = True if literal > 0 else False
        new_clauses = removeLiteralFromClauses(clauses, literal)
    else:
        new_clauses = clauses
    if not new_clauses:
        return True
    if [] in new_clauses:
        return False
    new_literal = chooseLiteral(new_clauses, unit_clauses)
    if new_literal:
        if DPLL(new_clauses, -new_literal, variable_values, unit_clauses):
            return True
        return DPLL(new_clauses, new_literal, variable_values, unit_clauses)
    return True
```

- First, if a literal is provided, its truth value is assigned in *variable_values*. Then, the function **removeLiteralFromClauses** is called to eliminate all clauses containing the assigned literal. This step reduces the search space and helps in simplifying the problem.
- If literal is **None**, it means the function is at the beginning of the search, and no literal has been assigned yet
- If *new_clauses* becomes empty after clause reduction, it means all clauses have been satisfied, and the function returns **True**, indicating a solution has been found.
- If an empty list [] is present in *new_clauses*, it indicates a contradiction (e.g., conflicting assignments), so the function returns **False**.
- The **chooseLiteral** function is to called to select new literal.
- If a new literal *new_literal* is found, the function recursively calls itself twice:
  - Once with the negation of *new_literal*, representing the case where the literal is assigned **False**.
  - Once with *new_literal* itself, representing the case where the literal is assigned **True**.
- If either of these recursive calls returns **True**, indicating that a solution has been found, the function immediately returns **True**.
- If both recursive calls return **False**, the search continues with other branches or backtracks if necessary.
- If no new literal is found (i.e., new_literal is **None**), the function returns **True** by default. This situation occurs when there are no more decisions to make, and the current partial assignment satisfies all remaining clauses.

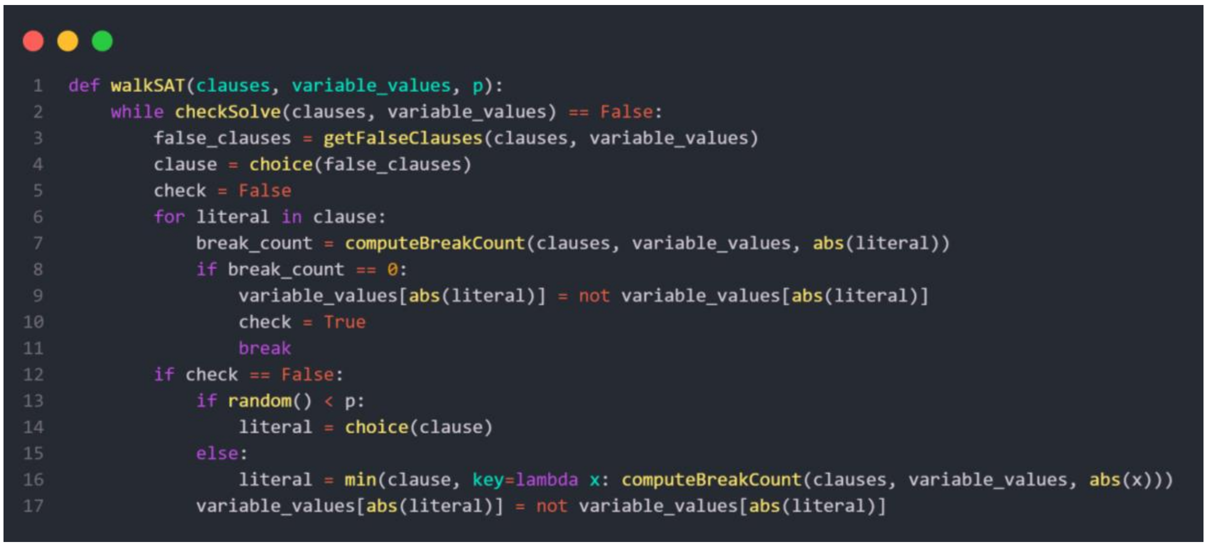
IV. WalkSAT

1. Idea

- The strategy for WalkSAT revolves around local search and random walk with enhancements to efficiently explore the solution space.
- The algorithm operates by randomly changing the truth value of a variable (switching it from **True** to **False** or vice versa).
- Each move's potential is evaluated using the break count, which measures the number of true clauses that become false due to the variable flip.
- It is worth noting that the make count, indicating the number of true clauses that become false, is generally less significant than the break count. This is because, as the algorithm approaches a solution, the make count tends to become negligible, while the break count remains crucial in determining whether a variable flip is beneficial or not.
- If the break count = 0, meaning it does not break any clauses, then this flip is probably beneficial and should be flipped.
- If there are no variable with break count = 0 then it probabilistically accepts the flip based on a parameter known as the probability **p**.
- This probabilistic acceptance allows the algorithm to explore the solution space by occasionally accepting moves that worsen the solution, akin to the exploration phase in simulated annealing.

2. Implementation

- Minor adjustments:
  - **assign_variables**: The same as bruteforce
  - **generateCNFFromConstraintsByCell** and **generateCNFFromConstraints**: The same as DPLL
- **checkSolve:** Recycle **removeLiteral** and **removeLiteralFromClauses** from DPLL, iterate over all variables and shorten overall set of clauses if resulting clause is [] then return **True**, else return **False**.
- **getFalseClauses:** Iterate over each clause, examine all of the literal in a specific clause, if all of the literals are **False** then append to a list. Return a list of clauses that are currently false.
- **computeBreakCount**: Computes the break count (with the above mentioned definition) of a literal in the context of unsatisfied clauses.
- **walkSAT:** Combine all of the above functions to find a solution to matrix.



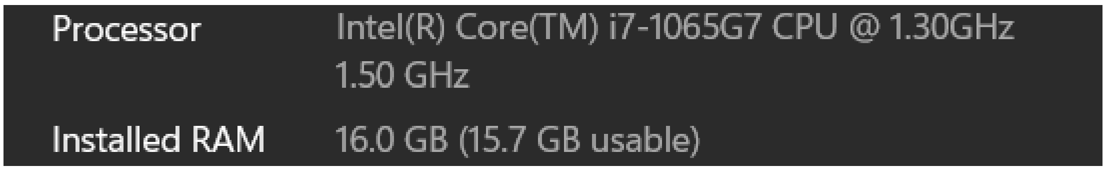
```python
def walkSAT(clauses, variable_values, p):
    while checkSolve(clauses, variable_values) == False:
        false_clauses = getFalseClauses(clauses, variable_values)
        clause = choice(false_clauses)
        check = False
        for literal in clause:
            break_count = computeBreakCount(clauses, variable_values, abs(literal))
            if break_count == 0:
                variable_values[abs(literal)] = not variable_values[abs(literal)]
                check = True
                break
        if check == False:
            if random() < p:
                literal = choice(clause)
            else:
                literal = min(clause, key=lambda x: computeBreakCount(clauses, variable_values, abs(x)))
            variable_values[abs(literal)] = not variable_values[abs(literal)]
```

- Loop Until Solved: Continuously iterate until all clauses are satisfied.
- Identify **False** Clauses: Within each iteration of the loop, the function first identifies the false clauses using the **getFalseClauses** function. These are the clauses that are not satisfied by the current variable assignments.
- Choose a Clause: Randomly select one of the unsatisfied clauses.
- Explore Variables in the Clause: Iterate through each literal within the chosen clause while also calculating the break count for each literal, measuring the number of true clauses that would become false if the literal's value were flipped.
- Flip a Literal: If a literal with a break count of 0 exists (meaning flipping it would not cause any additional clauses to become false), flip its truth value to satisfy the clause.
- Random or Greedy Choice: If no literal with a break count of 0 is found, randomly select a literal with probability p, or choose the one that minimizes the break count among all literals in the clause.
- Flip the Chosen Literal: Flip the truth value of the chosen literal and repeat the process until all clauses are satisfied or a termination condition is met.

G. EVALUATION & COMMENTS

I. Running device’s specifications

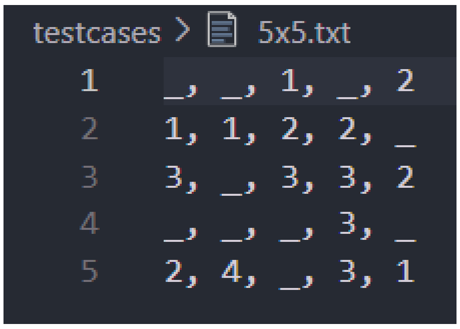
- Experiments are carried out on a laptop with the below specifications:



Processor	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Installed RAM	16.0 GB (15.7 GB usable)

II. Featured input

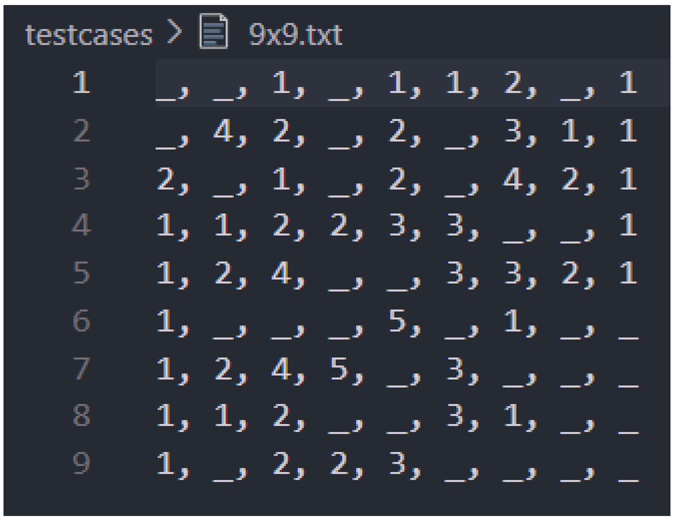
- 5x5 matrix:



testcases > 5x5.txt

```
_, _, 1, _, 2
1, 1, 2, 2, _
3, _, 3, 3, 2
_, _, _, 3, _
2, 4, _, 3, 1
```

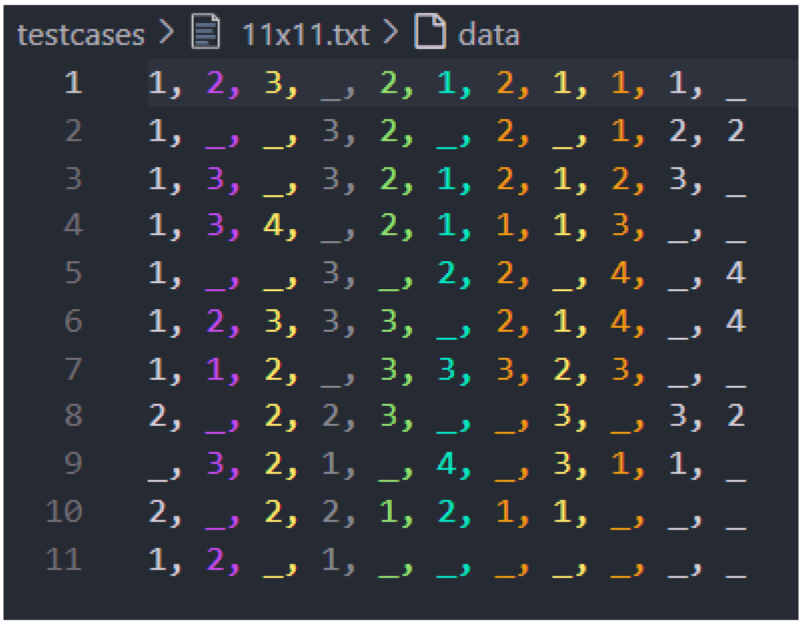
- 9x9 matrix:



testcases > 9x9.txt

```
_, _, 1, _, 1, 1, 2, _, 1
_, 4, 2, _, 2, _, 3, 1, 1
2, _, 1, _, 2, _, 4, 2, 1
1, 1, 2, 2, 3, 3, _, _, 1
1, 2, 4, _, _, 3, 3, 2, 1
1, _, _, _, 5, _, 1, _, _
1, 2, 4, 5, _, 3, _, _, _
1, 1, 2, _, _, 3, 1, _, _
1, _, 2, 2, 3, _, _, _, _
```

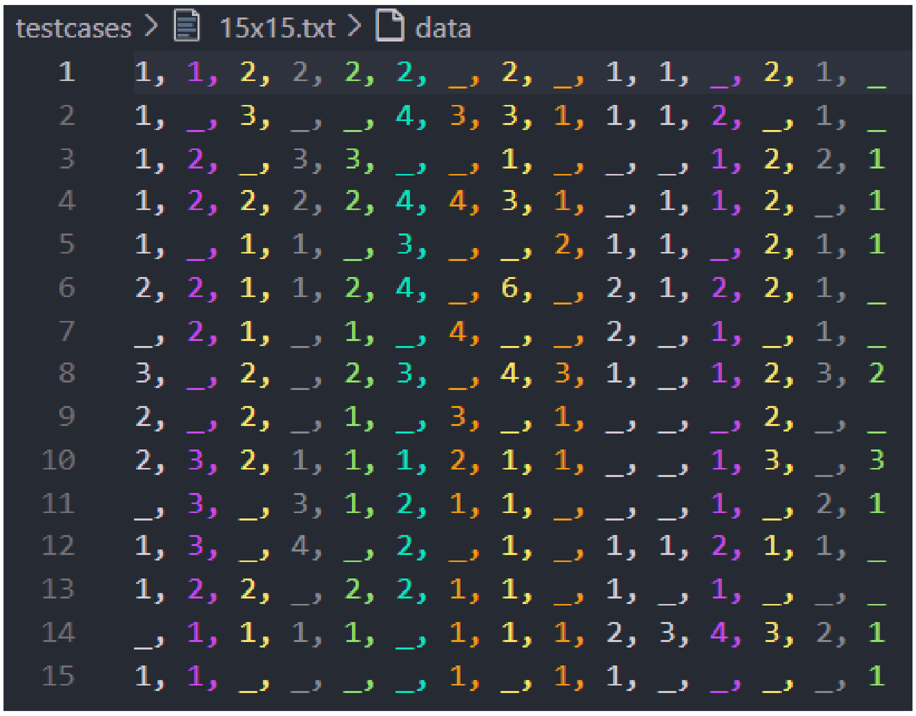
- 11x11 matrix:



testcases > 11x11.txt > data

```
1, 2, 3, _, 2, 1, 2, 1, 1, 1, _
1, _, _, 3, 2, _, 2, _, 1, 2, 2
1, 3, _, 3, 2, 1, 2, 1, 2, 3, _
1, 3, 4, _, 2, 1, 1, 1, 3, _, _
1, _, _, 3, _, 2, 2, _, 4, _, 4
1, 2, 3, 3, 3, _, 2, 1, 4, _, 4
1, 1, 2, _, 3, 3, 3, 2, 3, _, _
2, _, 2, 2, 3, _, _, 3, _, 3, 2
_, 3, 2, 1, _, 4, _, 3, 1, 1, _
2, _, 2, 2, 1, 2, 1, 1, _, _, _
1, 2, _, 1, _, _, _, _, _, _, _
```

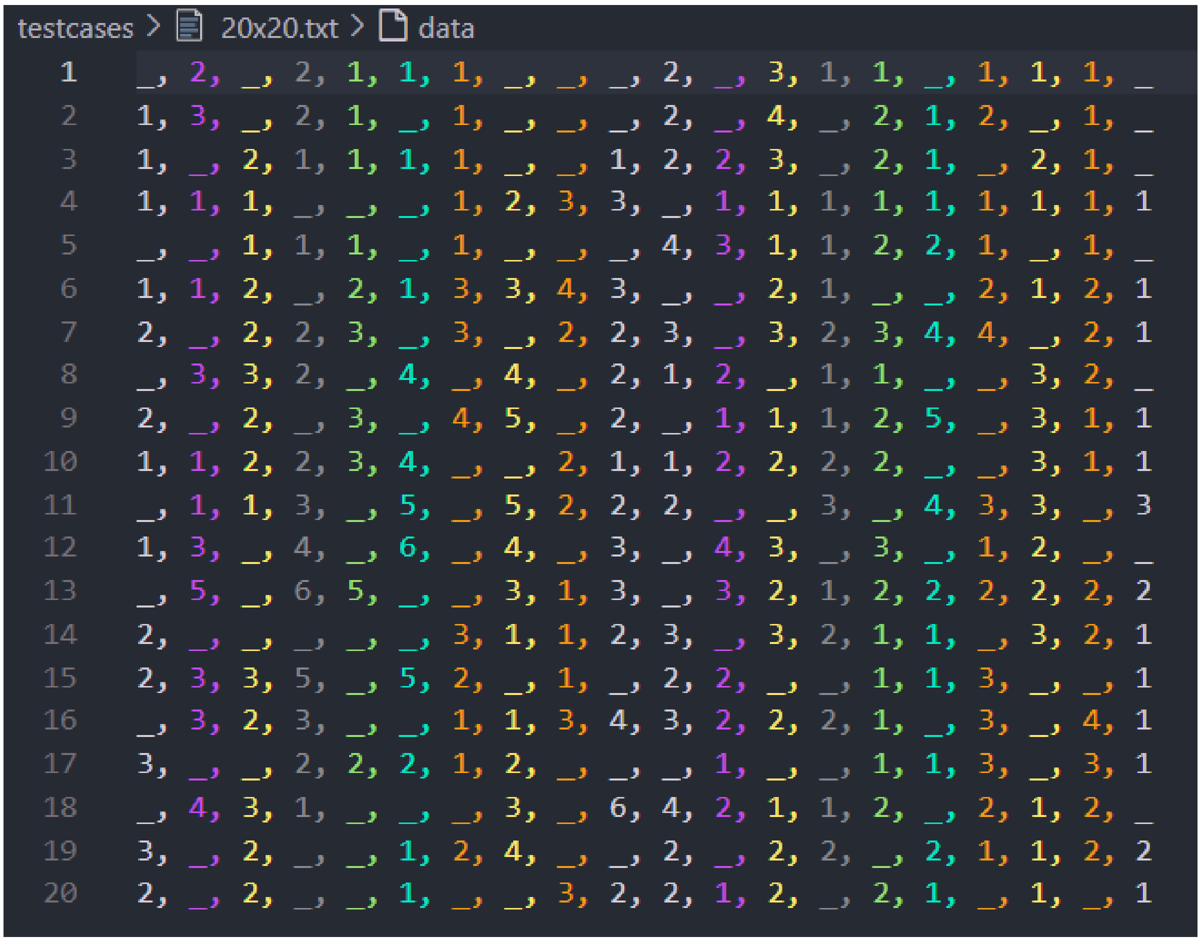
- 15x15 matrix:



testcases > 15x15.txt > data

```
1, 1, 2, 2, 2, 2, _, 2, _, 1, 1, _, 2, 1, _
1, _, 3, _, _, 4, 3, 3, 1, 1, 1, 2, _, 1, _
1, 2, _, 3, 3, _, _, 1, _, _, _, 1, 2, 2, 1
1, 2, 2, 2, 2, 4, 4, 3, 1, _, 1, 1, 2, _, 1
1, _, 1, 1, _, 3, _, _, 2, 1, 1, _, 2, 1, 1
2, 2, 1, 1, 2, 4, _, 6, _, 2, 1, 2, 2, 1, _
_, 2, 1, _, 1, _, 4, _, _, 2, _, 1, _, 1, _
3, _, 2, _, 2, 3, _, 4, 3, 1, _, 1, 2, 3, 2
2, _, 2, _, 1, _, 3, _, 1, _, _, _, 2, _, _
2, 3, 2, 1, 1, 1, 2, 1, 1, _, _, 1, 3, _, 3
_, 3, _, 3, 1, 2, 1, 1, _, _, _, 1, _, 2, 1
1, 3, _, 4, _, 2, _, 1, _, 1, 1, 2, 1, 1, _
1, 2, 2, _, 2, 2, 1, 1, _, 1, _, 1, _, _, _
_, 1, 1, 1, 1, _, 1, 1, 1, 2, 3, 4, 3, 2, 1
1, 1, _, _, _, _, 1, _, 1, 1, _, _, _, _, 1
```

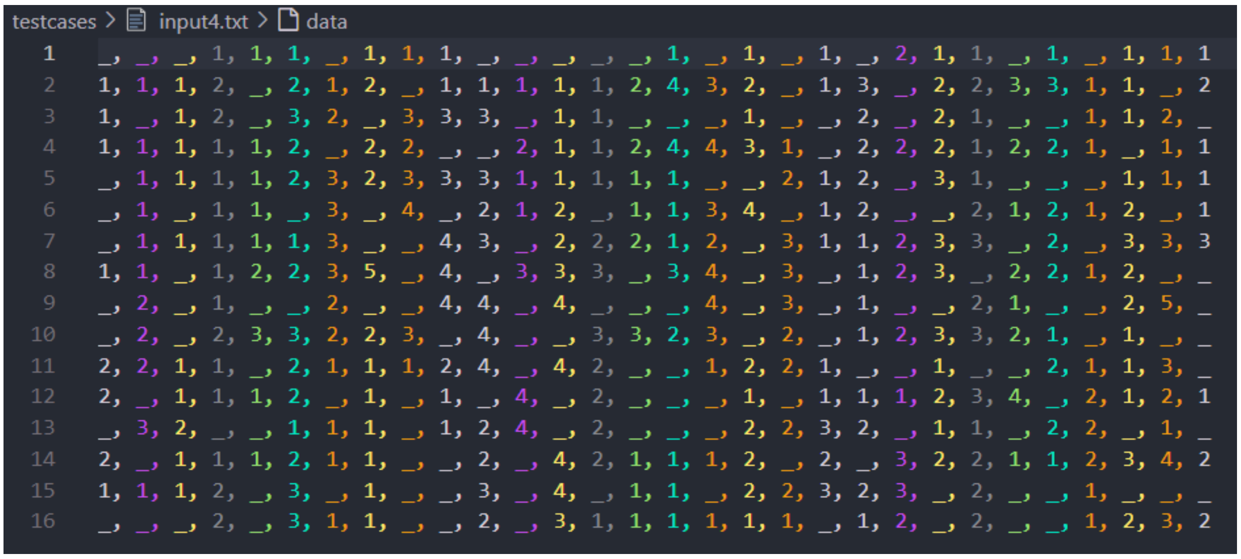
- 20x20 matrix:



testcases > 20x20.txt > data

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

- 16x30 matrix:


testcases > input4.txt > data

```
_, _, _, 1, 1, 1, _, 1, 1, 1, _, _, _, _, _, 1, _, 1, _, 1, _, 2, 1, 1, _, 1, _, 1, 1, 1
1, 1, 1, 2, _, 2, 1, 2, _, 1, 1, 1, 1, 1, 2, 4, 3, 2, _, 1, 3, _, 2, 2, 3, 3, 1, 1, _, 2
1, _, 1, 2, _, 3, 2, _, 3, 3, 3, _, 1, 1, _, _, _, 1, _, _, 2, _, 2, 1, _, _, 1, 1, 2, _
1, 1, 1, 1, 1, 2, _, 2, 2, _, _, 2, 1, 1, 2, 4, 4, 3, 1, _, 2, 2, 2, 1, 2, 2, 1, _, 1, 1
_, 1, 1, 1, 1, 2, 3, 2, 3, 3, 3, 1, 1, 1, 1, 1, _, _, 2, 1, 2, _, 3, 1, _, _, _, 1, 1, 1
_, 1, _, 1, 1, _, 3, _, 4, _, 2, 1, 2, _, 1, 1, 3, 4, _, 1, 2, _, _, 2, 1, 2, 1, 2, _, 1
_, 1, 1, 1, 1, 1, 3, _, _, 4, 3, _, 2, 2, 2, 1, 2, _, 3, 1, 1, 2, 3, 3, _, 2, _, 3, 3, 3
1, 1, _, 1, 2, 2, 3, 5, _, 4, _, 3, 3, 3, _, 3, 4, _, 3, _, 1, 2, 3, _, 2, 2, 1, 2, _, _
_, 2, _, 1, _, _, 2, _, _, 4, 4, _, 4, _, _, _, 4, _, 3, _, 1, _, _, 2, 1, _, _, 2, 5, _
_, 2, _, 2, 3, 3, 2, 2, 3, _, 4, _, _, 3, 3, 2, 3, _, 2, _, 1, 2, 3, 3, 2, 1, _, 1, _, _
2, 2, 1, 1, _, 2, 1, 1, 1, 2, 4, _, 4, 2, _, _, 1, 2, 2, 1, _, _, 1, _, _, 2, 1, 1, 3, _
2, _, 1, 1, 1, 2, _, 1, _, 1, _, 4, _, 2, _, _, _, 1, _, 1, 1, 1, 2, 3, 4, _, 2, 1, 2, 1
_, 3, 2, _, _, 1, 1, 1, _, 1, 2, 4, _, 2, _, _, _, 2, 2, 3, 2, _, 1, 1, _, 2, 2, _, 1, _
2, _, 1, 1, 1, 2, 1, 1, _, _, 2, _, 4, 2, 1, 1, 1, 2, _, 2, _, 3, 2, 2, 1, 1, 2, 3, 4, 2
1, 1, 1, 2, _, 3, _, 1, _, _, 3, _, 4, _, 1, 1, _, 2, 2, 3, 2, 3, _, 2, _, _, 1, _, _, _
_, _, _, 2, _, 3, 1, 1, _, _, 2, _, 3, 1, 1, 1, 1, 1, 1, _, 1, 2, _, 2, _, _, 1, 2, 3, 2
```

III. Experiment results

- **Note**: All results are the average of at least 5 runs. Instances of NaN are the result of waiting for 30+ minutes without an answer.

	Python-sat	Bruteforce	DPLL	WalkSAT
5x5	0.01315	0.01385	0.00874	0.00994
9x9	0.01524	0.01336	0.02038	0.01750
11x11	0.01653	0.01843	0.02524	0.02404
15x15	0.03241	1.07478	0.09514	0.08429
20x20	0.04372	0.04302	0.20211	0.13531
16x30	0.05208	NaN	0.33151	0.30633

IV. Comments

- For smaller puzzle sizes (e.g., 5x5 and 9x9), all algorithms seem to perform reasonably well, with relatively low execution times.
- However, as the puzzle size increases (e.g., 15x15, 20x20, and 16x30), the execution times of some algorithms, such as Brute Force and DPLL, exhibit significant increases, indicating scalability challenges. Bruteforce algorithm even fails to solve the 16x30 puzzle within acceptable time.
- Python-SAT appears to perform consistently well across different puzzle sizes, suggesting its efficiency in handling larger and more complex problem instances.
- Based on the observed performance, it is recommended to prioritize the use of python-sat and WalkSAT algorithms for Gem Hunter problem instances, particularly those of larger sizes.
- Further experimentation and analysis, including parameter tuning and

algorithmic optimizations, may provide insights into improving the efficiency and scalability of existing algorithms, especially DPLL, for larger problem instances.

H.ASSIGNMENT PLAN

Student ID	Full name	Assigned task	Completion evaluation	Total contribution
22127057	Đỗ Phan Tuấn Đạt	- Implemented walkSAT algorithm and main program UI. - Reported on the corresponding algorithm. - Combined and designed the group's report.	100%	25%
22127064	Phạm Thành Đạt	- Implemented DPLL algorithm. - Carried out experiment to record runtime of each algorithm. - Report on the corresponding sessions.	100%	25%
22127123	Lê Hồ Phi Hoàng	- Implemented Bruteforce algorithm. - Commented on the experiment results. - Report on the corresponding sessions.	100%	25%
22127131	Trần Nguyễn Minh Hoàng	- Implemented CNF generation and Python-sat solving algorithm. - Reported on the	100%	25%

		corresponding sessions. - Proof-read the group’s report.		

I. ESTIMATED COMPLETION

No.	Specifications	Scores	Estimated completion
1	**Solution description:** Describe the correct logical principles for generating CNFs	30%	100%
2	Generate CNFs automatically	10%	100%
3	Use pysat library to solve CNFs correctly	10%	100%
4	Implement an optimal algorithm to solve CNFs without a library	10%	100%
5	Program brute-force algorithm to compare with your chosen algorithm(speed) Program backtracking algorithm to compare with your chosen algorithm (speed)	10%	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	30%	100%

J. REFERENCES

- https://www.youtube.com/watch?v=z_JRErOYDso
- https://www.youtube.com/watch?v=xFpndTg7ZqA
- https://iiis.tsinghua.edu.cn/uploadfile/2015/1022/20151022155124653.pdf?fbclid=IwZXh0bgNhZW0CMTAAAR0KDbeoXZ-lTcLzX5UbFQduviX5qn0AAp7H--_9zdI_Xc_iVPHm-NucWtM_aem_Afj9mt4QKeMz0f9W_H6urZTjtKMrAwOPfCC22wuBspog027-JQd4xig6l57l7rQS-5YxfUfbxGn9o6l8GaSq9QWS
- https://davidnhill.github.io/JSMinesweeper/index.html?board=30x16x99&f

bclid=IwZXh0bgNhZW0CMTAAAR1J9db9lApPx70kg0Z4twn28jOptpyGS_
2nSPakgzICOxR3kk-
xdnUD9lk_aem_ATZwNtuJMvKFH7DNz2sD4KIeR2WgDdmcISmCu32xe
878Os-rRqriLvvdfNpTFIrXcV3EApMZP87pG_OcEI93MQWO