A magnifying glass and light bulb

Description automatically generated

1. **GENERAL TO THE PROBLEM**
2. **Description of Minesweeper**

**A screenshot of a game

Description automatically generated**

1. **Requirement**

|  |  |  |
| --- | --- | --- |
| **No.** | **Criteria** | **Scores** |
| **1** | Solution description: Describe the correct logical principles for generating CNFs. | **30%** |
| **2** | Generate CNFs automatically | **10%** |
| **3** | Use pysat library to solve CNFs correctly | **10%** |
| **4** | Implement A\* to solve CNFs without a library | **10%** |
| **5** | Program brute-force algorithm to compare with A\* (speed)  Program backtracking algorithm to compare with A\* (speed) | **10%** |
| **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%** |

1. **Team members introduction**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Student’ID** | **Full-Name** | **Email** |
| **1** | 21127003 | Phan Thanh An | ptan21@clc.fitus.edu.vn |
| **2** | 21127014 | Phạm Hồng Gia Bảo | phgbao21@clc.fitus.edu.vn |
| **3** | 21127228 | Nguyễn Gia Bảo | ngbao21@clc.fitus.edu.vn |
| **4** | 21127294 | Nguyễn Hi Hữu | nhhuu21@clc.fitus.edu.vn |

1. **CONTRIBUTION AND GRADING**
2. **Contribution**

Although this is the first time, we have worked in a group together, every member has co-operated effectively and successfully to finish the task achievement of this project. The following table describes what each of us has contributed to the whole project.

|  |  |  |
| --- | --- | --- |
| **Member** | **Task Achievement** | **Contribution** |
| Phạm Hồng Gia Bảo | * Take responsibility for research and **implementation Brute Force.** * **Writing the Report Document.** | * Analyze, write reports as well as review the algorithms with other members. |
| Nguyễn Gia Bảo | * All together take responsibility for **Generate CNFs automatically and** * **Use pysat library to solve CNFs correctly.** | * Contribute very first ideas to develop this project. * Exchange knowledge with others |
| Nguyễn Hi Hữu | * All together take responsibility for **Implement A\* to solve CNFs without a library** | * Research and represent ideas on how to implement algorithms. |
| Phan Thanh An | * Take responsibility for **research and Program backtracking algorithm** | * Research on how to implement algorithms. * Edit and check all the knowledge written in the report. |

1. **Grading**

|  |  |  |
| --- | --- | --- |
| **No.** | **Criteria** | **Completion** |
| **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 A\* to solve CNFs without a library | **100%** |
| **5** | Program brute-force algorithm to compare with A\* (speed)  Program backtracking algorithm to compare with A\* (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%** |

1. **PRINCIPLES FOR GENERATING CNFs**

To generate CNFs (Conjunctive Normal Form) for solving minesweepers, you need to apply logical principles based on the rules and constraints of the game. Here are the correct logical principles to consider:

1. **Clue constraints**

* For each cell with a clue number (k) in the matrix, create a clause that states the number of adjacent mines must be equal to k. If there are “n” adjacent covered cells, the clause would be (C1 OR C2 OR ... OR Cn), where C1, C2, ..., Cn represent the covered cells adjacent to the clue cell. For example, if a cell has a clue of “2” and has three adjacent covered cells the clause would be (C1 OR C2 OR C3), where C1, C2, and C3 represent the covered cells adjacent to the clue cell.
* For example, let's consider a cell with a clue number of "2" and three adjacent covered cells (C1, C2, C3). The corresponding clause would be (C1 OR C2 OR C3). This clause ensures that at least two of the adjacent cells are mines, satisfying the clue number requirement.

1. **Non-mine constraints**

* For each covered cell in the matrix, create a clause that states the cell cannot contain a mine. This can be represented as the logical negation of the cell variable. For example, if C represents a covered cell, the non-mine constraint clause would be ¬C, where ¬ represents logical negation.
* For instance, if a cell contains a clue number, it implies that the cell itself cannot contain a mine. So, if C represents a cell with a clue number, the corresponding non-mine constraint clause would be ¬C.

1. **Mine constraints**

* For each mine cell in the matrix, create a clause that states the cell must contain a mine. This is represented as a positive literal, typically using a variable like M, where M represents the mine cell.
* For example, if a cell C1 has a clue of 3 and its only neighbors are C2, C3, and C4, then it can be concluded that C2, C3, and C4 are 100% mines. To represent this in CNF, you would create the clause (M2 OR M3 OR M4), where M2, M3, and M4 represent the mine cells corresponding to C2, C3, and C4 respectively.

1. **Exclusivity constraints (known as Adjacency Constraints)**

* For each pair of adjacent cells C1 and C2, create a clause that states if C1 has a mine, then C2 cannot have a mine, and vice versa. This is represented as (¬C1 OR ¬C2) to enforce the exclusivity between the two cells.
* For example, let's consider two adjacent cells C1 and C2. The corresponding exclusivity constraint clause would be (¬C1 OR ¬C2), which states that if C1 has a mine (¬C1 is true), then C2 cannot have a mine (¬C2 must be true), and vice versa.

1. **Uniqueness Constraints**

* For each set of cells that share a common neighbor, create a clause that ensures the uniqueness of mines among them. This clause states that only one cell in the set can contain a mine.
* For example, if cells B, C, and D have a common neighbor of cell A, and A has a clue of "1", mean that B, C, D cannot both contain more than 2 mines. This is represented as (¬B OR ¬C) AND (¬B OR ¬D) and (¬C OR ¬D).
* Also, if A has a clue is 2, mean that B, C, D cannot contain mines less than that clue number.

1. **Border Constraints**

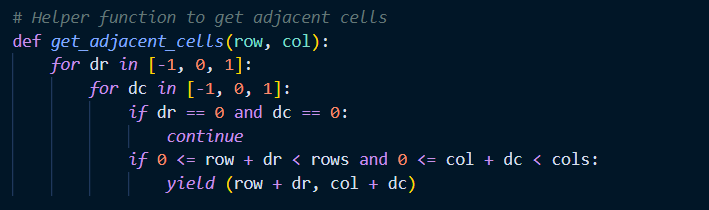
* For cells on the border of the matrix, adjust the constraints to consider the cells that might be outside the matrix as uncovered cells. This ensures that the adjacency and clue constraints are correctly applied.
* Let's consider a cell C on the border of the matrix. If C has a clue number that indicates adjacent mines, you need to account for the possibility of cells outside the matrix being uncovered. For example, if C is on the top-left corner of the matrix, you would adjust the adjacency constraint accordingly.
* For adjacent cells to the left and above C, you would consider the possibility of cells outside the matrix being uncovered. For instance, if C has a clue number of 2 and the adjacent cells are C1 and C2, you would create the following clause:
* This clause ensures that if C is a mine, either C1, C2, or both can't be mines. This adjustment accounts for the possibility of cells outside the matrix being uncovered.

1. **GENERATE CNFs AUTOMATICALLY**
2. Firstly, the **generate\_minesweeper\_CNF()** initializes variables for the CNF list (CNF), the number of rows and columns in the grid (rows and cols), and a set to store added clauses **added\_clauses**.

A close-up of text

Description automatically generated

1. ***def get\_adjacent\_cells(row, col):*** is a helper function that takes a cell's row and column as input and yields the coordinates of its adjacent cells (excluding the cell itself) using nested loops. It iterates over the possible offsets **dr and dc** within a 3x3 neighborhood centered around the given cell. It skips the case where both **dr and dc** are 0 to skip the current cell, and it checks if the resulting coordinates are within the grid boundaries before yielding them.



1. ***def add\_clause(clause):*** is a helper function that adds a clause to the CNF list if it has not already been added. It checks if the clause is already in the **added\_clauses** set to avoid duplications. If the clause is not present, it appends it to the CNF list and adds it to the **added\_clauses** set.

A computer code with white text

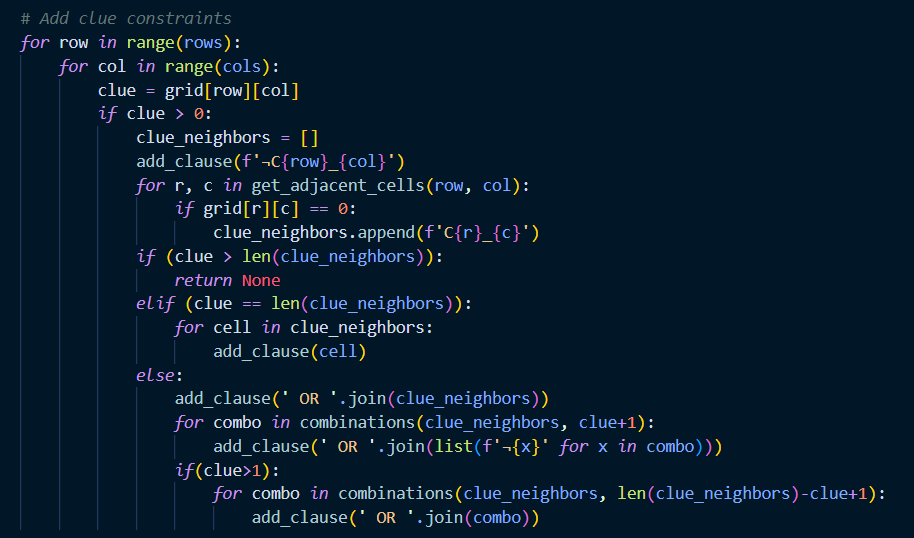
Description automatically generated

1. ***def is\_safe(row, col):*** is a helper function that takes the row and column of a cell as input and **checks if that cell is a "safe" cell** (where no adjacent cells contain mines). It does this by iterating over the adjacent cells using nested loops. If any of the adjacent cells contain a non-zero value (indicating a mine) or 'X' (indicating an unknown cell), it returns False. If none of the adjacent cells violate the safety condition, it returns True.

A computer screen with text

Description automatically generated

1. Then, we add clue constraints:
   * This section adds the clue constraints to the CNF. It iterates over each cell in the grid using nested loops.
   * If the clue value (grid[row][col]) of the current cell is greater than 0, it means the cell is a numbered cell and not an empty or mine cell.
   * It creates an empty list **clue\_neighbors** to store the adjacent cells that are currently unrevealed (with a value of 0).
   * It adds a clause to the CNFs for the negation of the current cell (¬C{row}\_{col}) indicating that the current cell is not a mine.
   * It then iterates over the adjacent cells using **get\_adjacent\_cells(row, col)** and checks if any of them are unrevealed cells (with a value of 0). If an adjacent cell is unrevealed, it adds its corresponding variable (C{r}\_{c}) to the **clue\_neighbors** list.
   * If the number of adjacent unrevealed cells **(len(clue\_neighbors))** is less than the clue value (clue), it means the clue cannot be satisfied, and it returns None to indicate that the CNF cannot be generated.
   * If the number of adjacent unrevealed cells is equal to the clue value, it means all adjacent cells are mines, so it adds a clause for each adjacent cell to the CNF.
   * If the number of adjacent unrevealed cells is greater than the clue value, it adds a clause that combines all the adjacent cells using the logical OR operator **(' OR '.join(clue\_neighbors))**.
   * It then generates additional clauses to handle various combinations of adjacent cells based on the clue value. These additional clauses ensure that the number of adjacent cells that contain mines matches the clue value.
   * If the clue value is greater than 1, it generates clauses for combinations of adjacent cells where the number of mines is equal to or greater than (clue + 1) and equal to or less **than (len(clue\_neighbors) - clue + 1).**

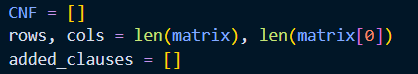


1. After that, we add safe cell constraints:
   * It iterates over each cell in the grid using nested loops.
   * If the value of the current cell is 0 (indicating an unrevealed cell) and the **is\_safe()** function returns True for that cell, it means the cell is safe (no adjacent cells contain mines).
   * It adds a clause for the variable corresponding to the safe cell **(C{row}\_{col})** to the CNF.

A screenshot of a computer program

Description automatically generated

1. Finally, the function returns the CNF list, which represents the constraints of the Minesweeper game.
2. **USE “PYSAT” LIBRARY TO SOLVE CNFs CORRECTLY**
3. **def generate\_minesweeper\_CNF(matrix):**
   * Firstly, we initialize the CNF list to store the CNF formula, rows and cols to store the dimensions of the matrix, and **added\_clauses** to keep track of the clauses that have been added to the CNF formula.



* + ***def get\_neighbors(row, col):*** takes a row and column index as input and returns a list of adjacent cells. It iterates over each adjacent cell by using nested loops with dr and dc representing the relative offsets. It checks if the adjacent cell is within the boundaries of the matrix and has a value of 0 (indicating an empty cell). If these conditions are met, it appends the index of the adjacent cell to the list.

A computer screen with text and symbols

Description automatically generated

* + ***def add\_clause(clause):*** takes a clause as input and adds it to the CNF formula if it hasn't already been added. It checks if the clause is not empty and if it is not already in the **added\_clauses** list. If these conditions are met, it appends the clause to the CNF list and adds it to the **added\_clauses** list.

A computer screen shot of a computer code

Description automatically generated

* + ***def is\_safe(row, col):*** checks if the cell with the coordinates (row, col) is safe. It iterates over all adjacent cells (excluding the cell itself) by using nested loops with dr and dc representing the relative offsets. It checks if the adjacent cell is within the boundaries of the matrix. If the value of the adjacent cell is not equal to 0, it means there is a mine adjacent to the current cell, and it returns False. If all adjacent cells have a value of 0, it means the current cell is safe, and it returns True.

A computer screen with text

Description automatically generated

* + ***After that,*** iterates over each cell in the matrix using nested loops with row and col representing the current cell's position. It retrieves the clue value of the current cell. If the **clue value is** **greater than 0**:
    - It adds a clause that the cell is not a mine to the CNF formula by appending [-(row \* cols + col + 1)] to the CNF list.
    - It calls the **get\_neighbors()** function to get a list of adjacent cells for the current cell and assigns it to **clue\_neighbors**.

A computer screen shot of text

Description automatically generated

1. **def solve\_minesweeper\_CNF(CNF, rows, cols,matrix):** 
   * Firstly, we create a solver using **Glucose3()** from the “**pysat library”**. It iterates over each clause in the CNF formula and adds them to the solver using the **add\_clause()** method.

A blue background with white text

Description automatically generated

* + We attempt to solve the CNF formula by calling the solve method. It returns a boolean value indicating whether a satisfying assignment was found (True) or not (False).

A blue background with white text

Description automatically generated

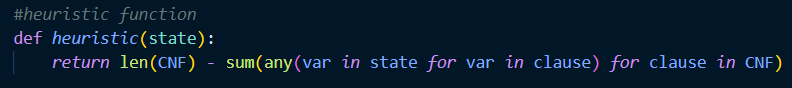
* + If the CNF formula can be solved (solution is True), it retrieves the satisfying assignment from the solver using the **get\_model** method. It initializes a new **solved\_matrix** with spaces (' ') for each cell. It iterates over each literal (lit) in the satisfying assignment. It calculates the corresponding row and column indices based on the literal value. If the value in the original matrix at that position is not 0, it assigns that value to the corresponding cell in the **solved\_matrix**. If the literal value is positive (lit > 0), it represents a mine, and 'X' is assigned to the corresponding cell. If the literal value is negative (lit < 0), it represents an empty cell, and '0' is assigned to the corresponding cell. Finally, it returns the **solved\_matrix** with the solution if it exists. Otherwise, it returns None.

A computer screen shot of a program code

Description automatically generated

1. **IMPLIMENTE A\* TO SOLVE CNFs WITHOUT A LIBRARY**
   * 1. **def generate\_minesweeper\_CNF(matrix)**
     2. **def solve\_minesweeper\_CNF(CNF, rows, cols,matrix,mine\_zone):**areimplemented like the way we solve problem using the pysat library.
     3. ***def A\_solve(CNF,mine\_zone):***

* ***def heuristic(state):*** Firstly, we define an inner function heuristic that takes a state as input. The heuristic function calculates a heuristic value for the given state. It counts the number of unsatisfied clauses in the CNF formula and subtracts it from the total number of clauses.



* ***def is\_goal(state):*** It checks if the state is a goal state by verifying if all clauses in the CNF formula are satisfied by the state.

A blue and white text

Description automatically generated

* ***def put\_mine(state,comb,list):*** It modifies the state by placing the mines from the combination in the corresponding cells in the list.

A computer screen shot of a code

Description automatically generated

* ***def generate\_successors(state,cell):*** It generates the successors of the given state by placing the mines in different combinations in the specified cell. It retrieves the list of cells and the number of mines from the **mine\_zone** for the given cell. It generates all possible combinations (comb) of the mines in the list with the given number of mines. For each combination, it creates a new successor state by copying the current state and placing the mines in the corresponding cells. It appends the new successor state to the list of successors.

A computer screen shot of code

Description automatically generated

* ***def optimize():*** The optimize function creates a valid initial state by initializing all variables with negative values. It determines the maximum variable index **num\_vars** in the CNF formula. It initializes the **initial\_state** as a list of negative values from **-1 to -(num\_vars)**. It also creates a list **flist** to store the variables that appear as unit clauses (clauses with a single literal). For each clause in the CNF formula, if the clause has a length of 1, it appends the absolute value of the literal to the **flist** and sets the corresponding variable in the initial state to the literal value. Finally, it returns the optimized initial state and the **flist.**

A computer code with colorful text

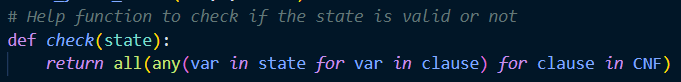
Description automatically generated

* Then, we continue A\_solve function continues with the main A\* algorithm implementation.
  + We initialize the open list as a priority **queue with** a single item: a tuple containing the heuristic value of the initial state, the step **count** (st) set to 0, and the initial state itself.
  + Then, we initialize the closed set as an empty set and add the tuple representation of the initial state to it.
  + The algorithm enters a loop that continues until the open list is empty.
  + In each iteration, it pops the item with the lowest heuristic value from the open list using **heapq.heappop**.
  + If the popped state is a goal state (**is\_goal(current\_state)** returns True), it converts the current state to a dictionary where the keys are the cell indices (+1) and the values are the mine numbers.
  + If the step count (st) is less than the number of cells in the mine zone (**len(mine\_zone)**), it generates the successors of the current state using **generate\_successors** for the current step count.
  + For each successor state, if the tuple representation of the state is not already in the closed set, it adds the tuple to the closed set, calculates the heuristic value for the state, and pushes the state onto the open list with the updated heuristic value and step count.
  + If the loop completes without finding a goal state, it returns None.

A computer screen shot of a program code

Description automatically generated

1. **IMPLEMENTATION OF BRUTE FORCE**
2. **def generate\_minesweeper\_CNF(matrix)** 
   * It is implemented like the way we solve problems using the pysat library.
3. **def solve\_minesweeper\_CNF(CNF, rows, cols,matrix,mine\_zone)**
   * It is implemented like the way we solve problems using the pysat library.
4. **def Brute\_force\_solve(CNF,n,unlist):**
   * Firstly, we define an inner function check that takes a state as input. The check function checks if the state satisfies all clauses in the CNF formula by verifying if each clause has at least one literal present in the state.



* + The code initializes a list **range\_check** with the range of variable indices from 1 to n. It also initializes the **init\_state** as a list of negative values from -1 to -n. For each variable in the **unlist**, it sets the corresponding variable in the **init\_state** to the value in the **unlist** and removes the variable from the **range\_check**.

A screen shot of a computer code

Description automatically generated

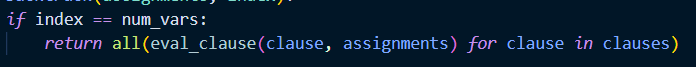
* + We set a loop that iterates **2n** times, representing all possible combinations of truth values for the remaining variables in **range\_check**. In each iteration, it creates a copy of the **init\_state** called state. For each variable in **range\_check**, it calculates the truth value based on the current iteration index i.
    - If the **j-th bit of i** is set (1), the variable j is assigned a positive value.
    - If the **j-th bit of i** is not set (0), the variable j is assigned a negative value.

It checks if the current state satisfies all the clauses in the CNF formula using the check function. If the **state is valid**, it **returns the state as a solution**. If the loop completes without finding a valid state, it returns None.

A computer code on a dark background

Description automatically generated

1. **IMPLEMENTATION OF BACKTRACKING**
2. **def generate\_minesweeper\_CNF(matrix)** 
   * It is implemented like the way we solve problems using the pysat library.
3. **def solve\_minesweeper\_CNF(CNF, rows, cols,matrix,mine\_zone)**
   * It is implemented like the way we solve problems using the pysat library.
4. **def solve\_cnf\_backtracking(clauses, num\_vars)**
   * ***def backtrack(assignments, index):*** is a recursive helper function that performs the backtracking algorithm
     + Firstly, we checks if the index has reached **num\_vars**, indicating that all variables have been assigned a value. If this condition is true, it returns the result of evaluating all clauses using the **eval\_clause()** function. It checks if all clauses are satisfied based on the current variable assignments.



* + - Then, we iterates over the possible values for the current variable (True and False). It assigns the value to assignments[index], recursively calls backtrack with the updated assignments and the next index, and checks if a satisfying assignment is found. If a satisfying assignment is found, it returns True. If not, it assigns None back to assignments[index] to backtrack and try the next value.

A computer screen shot of a code

Description automatically generated

* + - If all possible assignments have been tried for a variable and none of them lead to a satisfying assignment, the function returns False.
  + ***def eval\_clause(clause, assignments):*** represents a single clause in the CNF formula, and assignments, which contains the current variable assignments.
    - This line evaluates each literal in the clause based on the current variable assignments. If a positive literal (var > 0) is encountered, it retrieves the corresponding assignment (assignments[abs(var) - 1]). If a negative literal (var < 0) is encountered, it negates the corresponding assignment (not assignments[abs(var) - 1]). The any function is used to check if any of the literals evaluate to True. If at least one literal evaluates to True, the clause is considered satisfied.

A computer code with text

Description automatically generated with medium confidence

* + Here, the assignments list is initialized with None values for each variable. Then, the backtrack function is called with the initial assignments and index 0 to start the backtracking process. If a satisfying assignment is found (backtrack returns True), the function constructs a list of positive or negative integers representing the assigned values for each variable. The positive integers indicate True assignments, and the negative integers indicate False assignments. If no satisfying assignment is found, the function returns None.

A computer screen with colorful text

Description automatically generated

1. **EXPERIMENT AND REFLECTION, COMMENT**
2. **Experiment result**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Size of Matrix** | **3x3** | **5x5** | **7x7** | **9x9** | **15x15** |
| **A\*** | **0.000953** | **0.000985** | **0.001000** | **0.002063** | **Too long** |
| **Bruce Force** | **0** | **0.003976** | **0.042081** | **0.108667** | **Too long** |
| **Backtracking** | **; 0** | **0.003105** | **0** | **0.021997** | **Too long** |
| **PySat** | **0** | **0** | **0** | **0** | **0** |

**Note:** all the number above are measured in second (s)

1. **Reflection and Comment**

* **Pysat library:** This is a powerful Python library that provides a high-level interface for creating and solving Boolean satisfiability problems (SAT). By representing Minesweeper as a SAT problem, you can leverage PySAT's functionality to generate CNF formulas and find solutions efficiently. However, when using PySAT for Minesweeper problem solving, representing a Minesweeper instance as a SAT problem can lead to an exponential increase in the size of the CNF formula as the grid size grows. Therefore, it's important that we have to consider the trade-off between problem size and computational resources.

**=> It’s the great choice and powerful tool for generating CNFs and solving Minesweeper.**

* **A\* Implementation:** the performance of the code can be influenced by the complexity of the CNF formula, the size of the mine zone, and the effectiveness of the heuristic function.
  + **Heuristic Function:** The given heuristic function seems to be a simple estimate based on the number of unsatisfied clauses. Depending on the complexity of the CNF formula and mine zone, a more sophisticated heuristic function might be required to improve performance.
  + **Optimization:** The code includes an optimization step to initialize the initial state and lock unchangeable cells. This optimization can help reduce the search space and improve performance, especially when there are fixed cells in the CNF formula. However, the impact of this optimization may vary depending on the specific CNF and mine zone configurations.
* **Bruce Force Implementation**: it exhaustively searches through all possible truth value combinations for the remaining variables in the CNF formula to find a valid state that satisfies all the clauses. It starts with an initial state that sets the unchangeable variables and then iterates through all possible assignments for the remaining variables. If a valid state is found, it is returned as a solution. This approach has exponential time complexity since it needs to check all possible combinations.

**=> making it inefficient for large CNF formulas or when the number of variables is large.**

* **Backtracking Implementation**: The code uses an assignments list to store the current variable assignments during the backtracking process. It initializes the list with None values and updates them during the recursive calls. Memory usage depends on the number of variables **num\_vars**, as the list needs to store the assignment for each variable. The code is used for solving CNF formulas.



**=> While it can handle small to moderate-sized formulas efficiently, it may encounter challenges with larger or more complex formulas due to the exponential time complexity**.



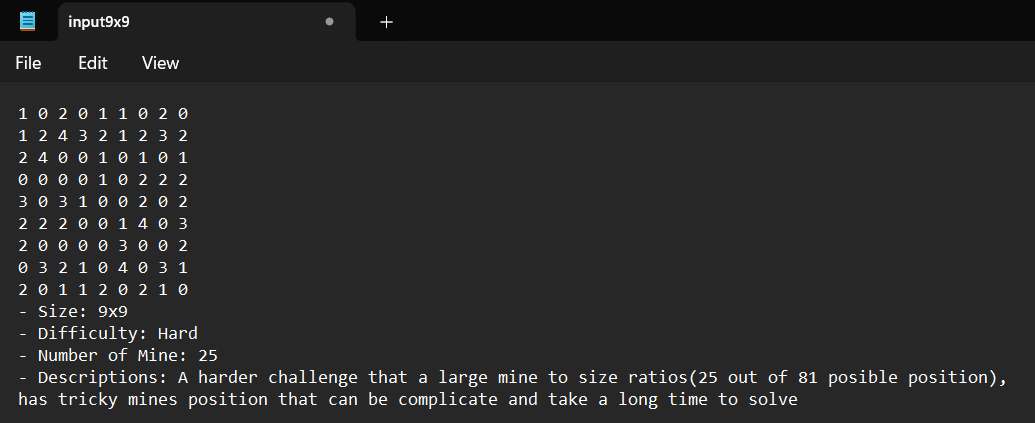
1. **USAGE AND DEMO**
   * 1. **Usage**

* **Step 1:** Choose and copy a test case in folder **“TEST CASE”.**

**A screenshot of a computer

Description automatically generated**



****

* **Step 2:** Open folder **“SOURCE”** in any source code editor.

**A black background with white text

Description automatically generated**



* **Step 3:** Paste that test case in file **“input.txt”**

**A screenshot of a computer

Description automatically generated**

* **Step 4:** Compile and run any algorithm you want.
* **Step 5:** View the solution at file **“ouput.txt”**

A screenshot of a computer

Description automatically generated

* + 1. **Demo**

**Link video demo:** [**https://youtu.be/lFJIw0gpOBw**](https://youtu.be/lFJIw0gpOBw)

1. **REFERENCE**
   * 1. Silde PowerPoint Mrs. Nguyen Ngoc Thao
     2. https://dspace.cvut.cz/bitstream/handle/10467/68632/F3-BP-2017-Cicvarek-Jan-Algorithms%20for%20Minesweeper%20Game%20Grid%20Generation.pdf?sequence=-1&isAllowed=y
     3. https://pysathq.github.io/docs/html/api/solvers.html