VIETNAM NATIONAL UNIVERSITY
HO CHI MINH UNIVERSITY OF SCIENCE

PROJECT 2 – MINE SWEEPER

ARTIFICIAL INTELLIGENCE

21CLC07

1. Trịnh Hạnh 21127040
2. Nguyễn Nhật Truyền 21127191
3. Trần Đỗ Anh Khoa 21127321
4. Nguyễn Khánh Hoàng 21127612

Contents

1 MEMBER INFORMATION

No.	Student ID	Full name
1	21127040	Trịnh Hạnh
2	21127191	Nguyễn Nhật Truyền
3	21127321	Trần Đỗ Anh Khoa
4	21127612	Nguyễn Khánh Hoàng

2 ASSIGNMENT

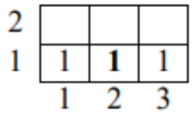
No.	Task	Member
1	Find solution for the problem	All
2	Handle file I/O of the program	Nguyễn Nhật Truyền
3	Convert board into CNF(s) clause	Trần Đỗ Anh Khoa Nguyễn Khánh Hoàng
4	Implent auto generated CNF	Trần Đỗ Anh Khoa
5	Use pySAT to solve CNF	Trần Đỗ Anh Khoa
6	Solve CNF using Genetic algorithm	Nguyễn Khánh Hoàng
7	Solve CNF using back-tracking	Nguyễn Nhật Truyền
8	Solve CNF using brute-force	Trịnh Hạnh
9	Generate test case	Nguyễn Khánh Hoàng Trịnh Hạnh
10	Testing	All
11	Write report	Trần Đỗ Anh Khoa Nguyễn Khánh Hoàng

3 COMPLETENESS

Completeness	Task	Notes
100%	Logical principles for generating CNF(s)	
100%	Generate CNF(s) automatically	
100%	Use pySAT to solve CNF	
100%	Solve CNF without library	In some case, time to complete is not consistent (due to the random specification in GA)
100%	Program brute-force, backtracking algorithm	Exponential time
100%	Comparison	

4 GENERATE CNF FROM A GIVEN BOARD

- In this section, we will describe how to generate CNF clauses from the given information. Note that if the variable is positive, this means this variable contain mine, otherwise this is a safe grid. Consider a board with given information:



- Consider grid [2,1], with this given board, we know that there are 5 adjacent, one of which contains a mine, and 2 adjacent have been explore. Let's n is the number of unknown adjacent, and k is the number of mine left, the grid [2,1] will have n=5-2=3 and k=1-0=1. And we will construct CNF clause based on two values n, k, call this proposition

$$KN(k, n)$$

- CNF requires a conjunction of disjunctions (representing as "at least one…") of literals. The literals can be either [I, j] contains a mine or [I, j] doesn't contain a mine. So, we can write KN (n, k) as two inequalities:

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

- With U (k, n) means that at most k of the n grids contain a mine and L (k, n) means that at least k of the n grids contains a mine.

- Consider U (k, n): at most k of the n grids contain a mine, we note that any subset of k+1 grids from the n unknown squares. If at most k are mines, then at least one is not a mine. So, we have the following by converse this statement:

$$U(k, n)\text{: for any } k+1 \text{ squares out of } n, \text{ at least one is not contain mine}$$

- Similarly, we can convert L (k, n) into for any n-k+1 grid out of n, at least one is a mine.

- By converting both inequalities, we now can easily construct U (k, n) by generate all combinations of selecting k+1 of n grids, this k+1 grids will be safe grids, and L(k, n) can be generated by create all combinations by select n-k+1 grids out of n, this grids will contain mine.

- With the above example:

$$U(k, n) \equiv (\neg X1,2 \vee \neg X2,2) \wedge (\neg X2,2 \vee \neg X3,2) \wedge (\neg X3,2 \vee \neg X1,2) \text{ and}$$

$$L(k, n) \equiv (X1,2 \vee X2,2 \vee X3,2) \text{ and}$$

$$KN(k, n) \equiv (\neg X1,2 \vee \neg X2,2) \wedge (\neg X2,2 \vee \neg X3,2) \wedge (\neg X3,2 \vee \neg X1,2) \wedge (X1,2 \vee X2,2 \vee X3,2)$$

- Note that there are two exceptions when k=0 and k+1>n. In case k=0, which means there are no more mine nearby, all variables will assign as negative value (represent safe grid). In case when k+1>n, which can only arise if k=n, which mean all adjacent contain mine, all variables will assign as positive value (represent mine grid)

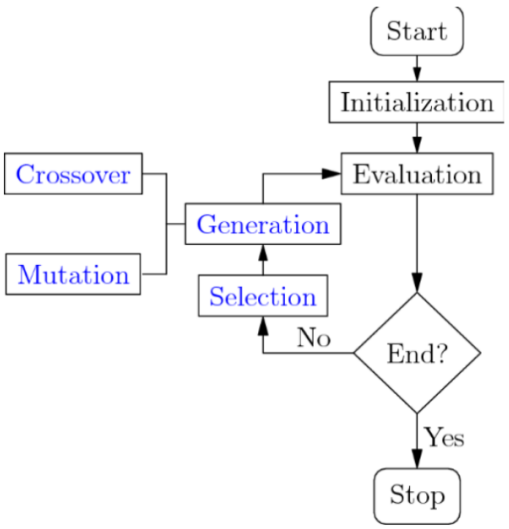
5 GENERATE CNF IN CODING ENVIRONMENT AND USE LIBRARY TO SOLVE CNF

- With the above conclusion, we can easily generate CNF by using combination built-in function in python. Loop through the board if the value of the grid, we will execute to generate CNF with the same logic as above section. And for the using pySAT purpose, we will represent Knowledge base (KB) contains all CNF (representing as a 1d-array) as a 2-d array, The KB will contain all CNF clause, which known as AND between each clause, and the CNF clause will contain all literals, which known as OR between each literal. And to simplify how data is represented, we will convert grid's index into position, which index [0][0] is at position 1, index [0][1] is at position 2, and so on (positive and negative value is at the same meaning as mentioned above). For instance, KB= [[1, -3, 5], [2, 3, -4]] can be known as" (1 or -3 or 5) and (2 or 3 or -4)"
- By creating KB like this, we can use pySAT to solve the problem without any standardization. In pySAT, we will use SAT solver named Glucose3. Add all clauses in the KB to the solver, and simply call solve function.

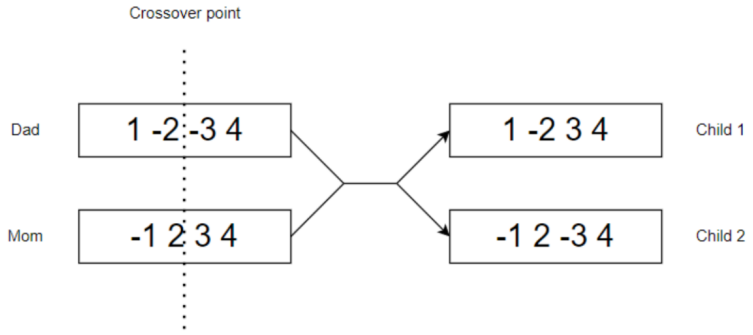
```
def pySAT(self):
    g = Glucose3()
    for clause in self.KB:
        g.add_clause (clause)
    if not g.solve():
        print('No solution')
    else:
        model = g.get_model()
        print(model)
```

6 SOLVE CNF WITHOUT USING LIBRARY

- To solve CNF (or SAT problem), there are various algorithms, especially modern SAT solver that can solve it quickly. But in this project, we decided to use Genetic algorithm, a superior local search algorithm to assign if the grid contains mine or not.



- For using this algorithm, we need to consider some aspect as follows:
  - **Encoding problem (chromosome):** To represent CNF as a chromosome, we simply store an array which shows the assignment of all variables, the way variables represent is similar to CNF. For example, a possible chromosome in the 2x2 board will be (1, -2, -3,4), which means there are no mines in position 2, 3, and mine is in 1, 4). And for accessing purpose, a chromosome will contain 'assignment' – assignment array and 'fitness' – point for evaluating how good this chromosome is.
  - **Fitness function:** To calculate the fitness function, we will use maximum heuristic, which heuristic is how many clauses is satisfied with this assignment. The higher satisfaction, the better the chromosome is. And if this assignment satisfies all clauses, which means this is one of the solutions.
  - **Selection:** In terms of creating new generations, we need to reproduction. And for this purpose, we use the roulette wheel to choose which chromosome will be taken to crossover and/or mutate. The probability for choosing chromosomes will depend on its fitness point. We simply use the equation "prob = fitness/totalfitness" to calculate the chance to choose this chromosome.
  - **Crossover:** After choosing the group of chromosomes that is the source of the next generation, we will execute crossover operation. For each pair to be mated, named dad and mom, a crossover point is chosen randomly from the position in the assignment. The first child will take from the first to crossover point of the dad, and the rest will take from the mom. In contrast, the second child will take from mom first, and then take the rest from dad.



– **Mutation:** To implement mutation operation, we need to hold a data represent mutate rate. In the algorithm, we will use random built-in function to generate a number which decides if this chromosome can mutate or not. In the mutation operation, we simply generate a random position in this assignment and change them to opposite value (assign=-assign)

By defining all those aspects, we will implement Genetic algorithm. Consider max generation before, after generating first generation (known as initial population), we will select, execute crossover and mutation operation chromosomes to generate next generation and then loop again. This loop will end whenever a chromosome is evaluated as goal state (satisfied with all clauses), which will return there is possible to solve, or reach max generation, which return there is no possible solution. In our program, we decided max generation will be 10000, as this problem is not generated so much clause that 10000 generation cannot handle it.

Because of Genetic algorithm's attribute, with each chromosome, we need to loop through all the clauses, which mean, the little KB is, the better performance it have. So, we've create a function that can reduce some unnecessary information. With each single literal, we can use it to reduce other clause based on logical rules, and also reduces itself. For example, given KB=[[1], [1, 2, 3], [-1, 4]], we can use clause [1] to transform the second clause into None (already satisfied) and third clause into [4] (note that we also need to store [1] somewhere for fully information)

7 SOLVE CNF USING BACK-TRACKING

- **is_satisfied(clause, assignment):** This function checks whether a given clause is satisfied by a particular assignment of truth values to variables. It iterates through the literals in the clause and checks if at least one literal is true based on the assignment.

```
def is_satisfied(clause, assignment):
    for literal in clause:
        var = abs(literal)
        if var in assignment:
            if literal > 0 and assignment[var]:
                return True
            elif literal < 0 and not assignment[var]:
                return True
    return False
```

- **all_clauses_satisfied(cnf_formula, assignment):** This function checks if all the clauses in the CNF formula are satisfied by the given assignment. It uses the is_satisfied function to check each clause.

```
def all_clauses_satisfied(cnf_formula, assignment):
    return all(is_satisfied (clause, assignment) for clause in cnf formula)
```

- **get_unassigned_variable(cnf_formula, assignment):** This function finds an unassigned variable in the CNF formula. It iterates through each clause and literal to find a variable that is not yet assigned in the assignment.

```
def get_unassigned_variable (cnf_formula, assignment):
    for clause in cnf_formula:
        for literal in clause:
            if abs(literal) not in assignment:
                return abs(literal)
    return None
```

- **backtrack_satisfy(cnf_formula, assignment):** This is the main backtracking function. It takes the CNF formula and an assignment as input. If all clauses are satisfied by the assignment, it returns the assignment. If not, it recursively tries different truth values for unassigned variables and backtracks if necessary.

```
def backtrack_satisfy (cnf_formula, assignment):
    if all_clauses_satisfied(cnf_formula, assignment):
        return assignment.copy()
    variable = get_unassigned_variable(cnf_formula, assignment)
    if variable is None:
        return None
    for value in [True, False]:
        assignment[variable] = value
        result = backtrack_satisfy(cnf_formula, assignment)
        if result:
            return result
        del assignment[variable]
    return None
```

- Here's how the backtracking algorithm generally works:
  - Check if all clauses are satisfied. If they are, return the current assignment.
  - If not all clauses are satisfied, find an unassigned variable.
  - Try assigning True to the variable and recursively call backtrack_satisfy.
  - If the recursive call returns a satisfying assignment, return it.
  - If not, backtrack by removing the assignment to the variable.
  - Try assigning False to the variable and recursively call backtrack_satisfy.
  - If the recursive call returns a satisfying assignment, return it.
  - If neither value for the variable leads to a satisfying assignment, backtrack further.

This implementation seems to be a straightforward implementation of the backtracking algorithm for solving SAT problems. However, the backtracking approach can become quite slow for large or complex CNF formulas due to its exponential time complexity in the worst case. There are more advanced techniques and algorithms that can improve the efficiency of solving SAT problems.

8 BRUTE FORCE ALGORITHM

- In order to implement the brute force algorithm, the idea is very simple: generate all possible assignments and then, check every assignment to find the one that satisfies all the CNF clauses.
- To generate all possible, we just use the positive assignment (all variables are positive), and replace the negative variable to create all assignments (2^(n*m)). To do this, we create all binary numbers of length n bits. Each binary number represents a sign change pattern for the elements in the original one

```
pool-[]
for i in range(2**n):
    new=[]
    for j in range(n):
        if(i>>j)&1:
            new.append(-assign[j])
        else:
            new.append(assign[j])
    pool.append(new)
```

- In terms of finding the solution, we reuse the isGoal() function in class Genetic algorithm (with a small modification). This function will return any assignment that satisfy all the clauses (which mean the solution)

9 COMPARISON

Because GA is not consistent in case of large CNFs, we will take average of 5 time running to compare with other algorithm

- Every test case that exceed 1000 second will consider as “exceed time”
- Instead of different size, we aim to the number of CNF, which mean we generate test case with the increasing number of CNF, not its size

Board	**Number of CNF**	**GA**	**Brute force**	**Back-tracking**
0 1 0 2 0 1 0 2 0	20	0.000	0.001000	0.002499
0 0 2 0 2 0 2 0 0 3 0 0 0 0 0 1	81	0.002	0.665001	1.202419
0 0 0 0 0 2 0 3 0 2 0 3 0 3 0 2 0 3 0 2 0 0 0 0 0	114	9.371	506.943	989.585257
0 1 0 0 0 0 1 0 6 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0	144	42.567	Exceed time	Exceed time
1 0 2 0 0 0 0 0 0 0 0 0 1 0 3 0 0 0 0 0 0 2 2 0 0 2 0 0 0 0 0 2 1 0 0 0	197	60.32	Exceed time	Exceed time