Undergraduate AI Capstone   NYCU Spr2023   Programming Assignment #3   Due 5/8/2023

In the lectures, logical inference techniques are only applied to tiny toy problems. The objective of this assignment is for you to solve a not-so-tiny problem automatically by using logical inference techniques explicitly. The target problem is the automatic playing of the Minesweeper game. (There have been a number of AI programs developed for Minesweeper in the web. However, this assignment will require you to use logical inference (specifically, the rule of **resolution**) explicitly so that you'll gain the experience. This is not the most efficient approach though. A typical approach will be more like a constraint satisfaction problem.)

For simplicity, you are only required to use **propositional logic** in your implementation. Treat each cell as a symbol that can be either false ("safe") or true ("mined"). The game proceeds as the player continues to mark the cells as safe or mined.

You program should consist of two main modules, one for game control and the other for the player.

Tasks of the game control module:

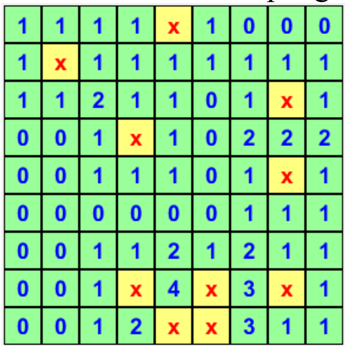
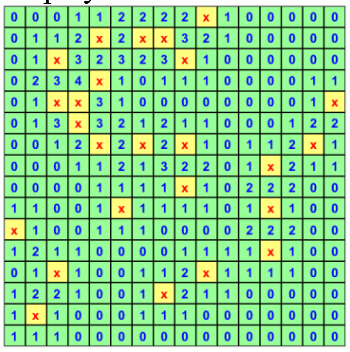
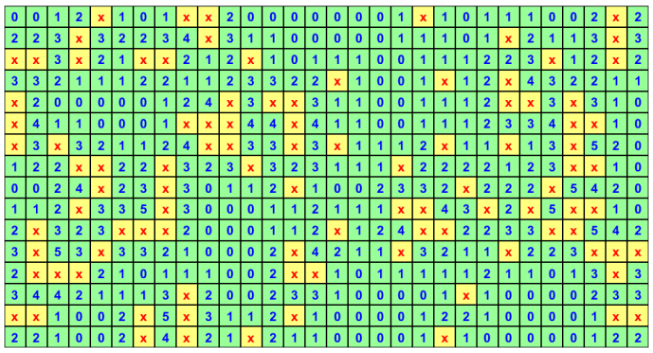
- Initialize the board with a preset number of mines. (For your reference, the three difficulty levels of the original Minesweeper are: Easy (9x9 board with 10 mines), Medium (16x16 board with 25 mines), and Hard (30x16 board with 99 mines).)
- Provide the hint (number of surrounding mines) when queried for a cell by the player module.
- Provide an initial list of "safe" cells.
    - The first few clicks of a Minesweeper game are usually random clicks intended to find connected regions of safe cells.
    - In this assignment, this is replaced by the initial safe cells.
    - To start, use round(sqrt(#cells)) as the number of initial safe cells. You can vary this number later to see how it affects the success rate of games.

Tasks of the player module:

- Here you need to maintain your KB containing CNF clauses.
- Also keep a list of single-lateral clauses that represent marked cells. This list is called KB0 below. The action of "marking a cell" is explained later.

When the game starts, the KB contains (negative) single-lateral clauses representing the initial safe cells. KB0 is empty initially.

Below are some sample game-play results:

* In the boards, gray cells are unmarked, green cells are those marked as safe, and yellow cells are those marked as mined. The numbers are hints of actually safe cells and the x's indicate actually mined cells.

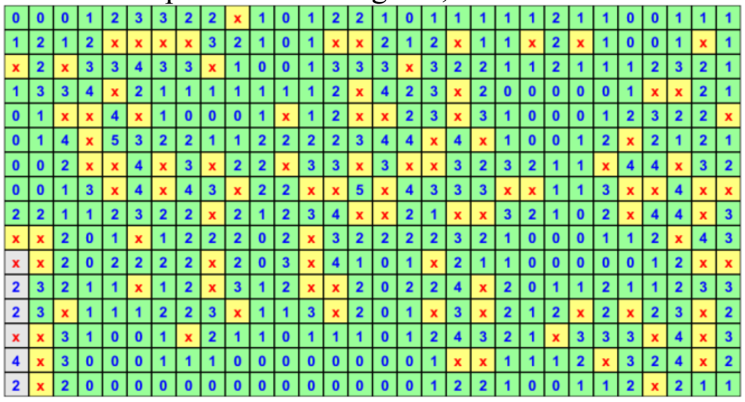
Game flow:

- When the game starts, the KB contains (negative) single-lateral clauses representing the initial safe cells. KB0 is empty initially.
- The game play proceeds in a loop. Within each iteration:

```
    If there is a single-lateral clause in the KB:
        Mark that cell as safe or mined.
        Move that clause to KB0.
```

Process the "matching" of that clause to all the remaining clauses in the KB.
If new clauses are generated due to resolution, insert them into the KB.
If this cell is safe:
Query the game control module for the hint at that cell.
Insert the clauses regarding its unmarked neighbors into the KB.
Otherwise:
Apply pairwise "matching" of the clauses in the KB.
If new clauses are generated due to resolution, insert them into the KB.
* For the step of pairwise matching, to keep the KB from growing too fast, only match clause pairs where one clause has only two literals.

Game termination:
- Success: When all the cells are marked. (KB should become empty.)
- From time to time, a game can be stuck in a state where no new markings can be made as there are multiple solutions to the unmarked cells. This commonly occurs when the difficulty level is Hard, especially near the end of the game.
- For this assignment, if several iterations pass without marking a new cell, you can just stop the game. The game is likely "stuck". (You can try to make guesses as in real games, but this is not required here.)
- Here is an example of a "stuck" game; there are 6 unmarked cells remaining (bottom-left corner).



0	0	0	1	2	3	3	2	2	x	1	0	1	2	2	1	0	1	1	1	1	1	2	1	1	0	0	1	1	1
1	2	1	2	x	x	x	x	3	2	1	0	1	x	x	2	1	2	x	1	1	x	2	x	1	0	0	1	x	1
x	2	x	3	3	4	3	3	x	1	0	0	1	3	3	3	x	3	2	2	1	1	2	1	1	1	2	3	2	1
1	3	3	4	x	2	1	1	1	1	1	1	1	2	x	4	2	3	x	2	0	0	0	0	0	1	x	x	2	1
0	1	x	x	4	x	1	0	0	0	1	x	1	2	x	x	2	3	x	3	1	0	0	0	1	2	3	2	2	x
0	1	4	x	5	3	2	2	1	1	2	2	2	2	3	4	4	x	4	x	1	0	0	1	2	x	2	1	2	1
0	0	2	x	x	4	x	3	x	2	2	x	3	3	x	3	x	x	3	2	3	2	1	1	x	4	4	x	3	2
0	0	1	3	x	4	x	4	3	x	2	2	x	x	5	x	4	3	3	3	x	x	1	1	3	x	x	4	x	x
2	2	1	1	2	3	2	2	x	2	1	2	3	4	x	x	2	1	x	x	3	2	1	0	2	x	4	4	x	3
x	x	2	0	1	x	1	2	2	2	0	2	x	3	2	2	2	2	3	2	1	0	0	0	1	1	2	x	4	3
x	x	2	0	2	2	2	2	x	2	0	3	x	4	1	0	1	x	2	1	1	0	0	0	0	0	1	2	x	x
2	3	2	1	1	x	1	2	x	3	1	2	x	x	2	0	2	2	4	x	2	0	1	1	2	1	1	2	3	3
2	3	x	1	1	1	2	2	3	x	1	1	3	x	2	0	1	x	3	x	2	1	2	x	2	x	2	3	x	2
x	x	3	1	0	0	1	x	2	1	1	0	1	1	1	0	1	2	4	3	2	1	x	3	3	3	x	4	x	3
4	x	3	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	x	x	1	1	1	2	x	3	2	4	x	2
2	x	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	2	1	0	0	1	1	2	x	2	1	1

About generating clauses from the hints:
- Each hint provides the following information: There are n mines in a list of m unmarked cells.
- (n == m): Insert the m single-literal positive clauses to the KB, one for each unmarked cell.
- (n == 0): Insert the m single-literal negative clauses to the KB, one for each unmarked cell.
- (m>n>0): General cases (need to generate CNF clauses and add them to the KB):
  C(m, m-n+1) clauses, each having m-n+1 positive literals
  C(m, n+1) clauses, each having n+1 negative literals.
  For example, for m=5 and n=2, let the cells be x1, x2, …, x5:
  There are C(5,4) all-positive-literal clauses:
  (x1 ∨ x2 ∨ x3 ∨ x4), (x1 ∨ x2 ∨ x3 ∨ x5), …, (x2 ∨ x3 ∨ x4 ∨ x5)
  There are C(5,3) all-negative-literal clauses:
  (¬x1 ∨ ¬x2 ∨ ¬x3), (¬x1 ∨ ¬x2 ∨ ¬x4), (¬x1 ∨ ¬x2 ∨ ¬x5), …, (¬x3 ∨ ¬x4 ∨ ¬x5)
- The global constraint (total number of mines) can be considered a hint as well. However, it is too large to use in the beginning (m=#cells, n=#mines). However, you can apply this hint when the total number of unmarked cells is small enough.

About inserting a new clause to the KB:
- Do resolution of the new clause with all the clauses in KB0, if applicable. Keep only the resulting clause.
- Skip the insertion if there is an identical clause in KB.
- Check for **subsumption** with all the clauses in KB:
  An example of subsumption:
  (x2 ∨ x3) is stricter than (x1 ∨ x2 ∨ x3): The former entails the latter.
  As a result, we do not need the less strict clause anymore.
  New clause is stricter than an existing clause): Delete the existing clause.

An existing clause is stricter than the new clause): Skip (no insertion).

About "matching" two clauses:

- Check for duplication or subsumption first. Keep only the more strict clause.
- If no duplication or subsumption, and they have complementary literals:

```
If there is only one pair of complementary literals:
    Apply resolution to generate a new clause, which will be inserted into the KB.
If there are more than one pairs of complementary literals:
    Do nothing here. (Resolution will results in tautology (always true).)
    Example: (¬x2 ∨ x3) and (x1 ∨ x2 ∨ ¬x3)
```

- **Unit-propagation** heuristic:

```
Focus on pairs involving single-literal clauses first in the resolution process to avoid KB explosion.
Example: (x1) and (¬x1 ∨ ¬x2).
The approach:
    Whenever we obtain a new single-literal clause A:
        For each multi-literal clause containing A:
            If the two occurrences of A are both positive or both negative:
                Discard the multi-literal clause. It is always true. (This is a case of subsumption.)
            Else
                Remove A from the multi-literal clause. This is the result of resolution.
    Whenever new clauses are generated from a new hint:
        Check against the list of single-literal clauses first.
```

- Using the unit-propagation heuristic leads to the maintenance of a state where no resolution can be applied between a single-literal clause and another clause.

Your submission should be a report file in PDF format. The report (maximum 10 pages single-spaced) should contain the sections listed above. Submit your report through E3. Late submission is accepted for up to 3 days, with a 10% deduction per day.

Include your program code as an appendix (not counting toward the 10-page limit), starting from a separate page. You can use C/C++ or Python to write your program. In general, the TAs will not actually compile or run your programs. The code listing is used to understand your thoughts during your implementation and to find problems if your results look strange. Therefore, the code listing should be well-organized and contain comments that help the readers understand your code; this will also affect your grade.