AI Prog3

109550087 單宇晟

**Game control module**

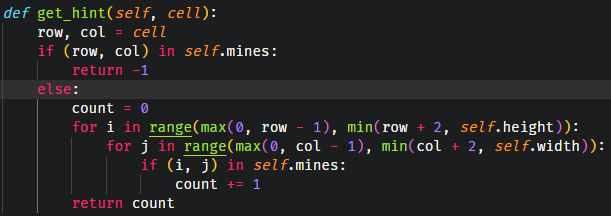
* Initialize

When the game starts, we can decide the difficulty of the game for AI to play. And this will initialize the board as the spec said.



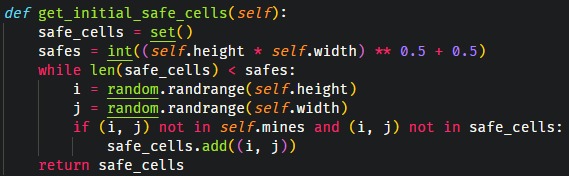
* Provide the hint

get\_hint() will provide each cell the number of its neighboring mines, which help us make clauses later.



* Provide an initial list of "safe" cells

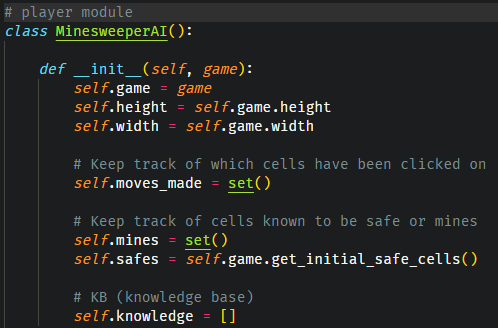
Since the first few clicks of a Minesweeper game are usually random clicks, to avoid losing at the first step, get\_initial\_safe\_cells() provide a list of some safe cells to click in the early game stage. The number of initial safe cells are round(sqrt(#cells)). Also, this provide an additional information of the game board, which makes AI start to gain knowledge from those cells.



**Player module**

* Initialize

Here, self.knowledge is the knowledge base (KB) in my program, it would save clauses which were gained during playing.

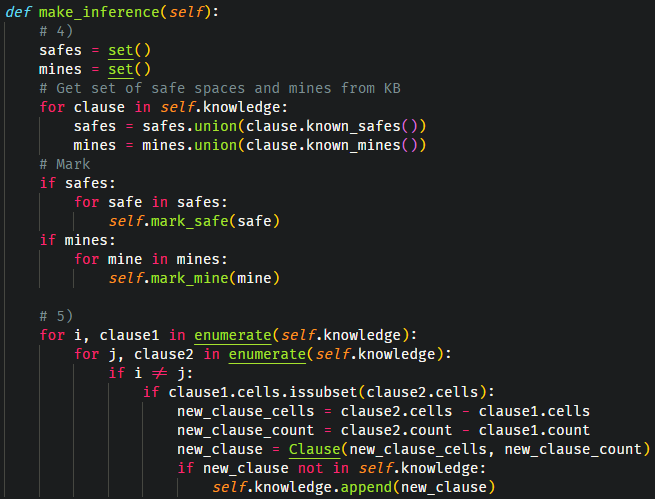


**Game flow**

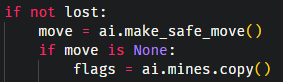
In general, we first make a safe move (the first move will be chosen from initial safe cells) and mark it as a safe cell. Next, I find the remaining neighbors which is not a mine, and make this clue into a new clause. If the new clause is a single-lateral clause, mark it as safe or mine.



After add\_knowledge(), I start to organize those clauses AI has so far, which is the “matching” part in the spec. In the beginning, I combine safe cells and mine among all clauses, mark them as safe or mine. Then I check if any two different clause has a “subset” relationship. If a clause is a subset of another, we can know that the rest of the larger clause (except the smaller clause part) can become another clause, too. Moreover, if the new clause we find hasn’t been in the knowledge base, add it into our knowledge base.

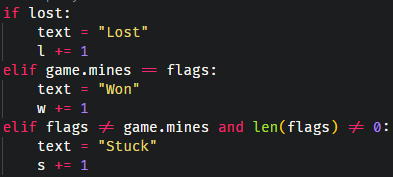


**Game termination**

While the game state is not lost, AI will keep making moves. However, if there are no more safe moves, save the mines’ location (flag), this help us decide the game state. 

Note that if there are no more safe moves, the situation itself has two possible conditions:

1. The remaining cells are all mines
2. The remaining cells may be safe or a mine, but AI can tell from the knowledge base.



As a result, if game.mines is the same as flags, which means that AI has mark all mines on the board, we can know that AI has won the game. On the other hand, if flags isn’t the same as game.mines, which means that AI hasn’t mark all mines on the board, and there is no more safe moves to make, the game becomes “stuck”.

**Result**

When the number of initial safe cells are round(sqrt(#cells)):

|  |  |  |  |
| --- | --- | --- | --- |
| Difficulty | easy | medium | hard |
| Win rate (%) | 90 | 95 | 10 |

Here, the result of medium is higher than easy mode. I think that this is because the density of mines (#mines / #cells) are higher in easy mode, which means that mines are more likely to be arranged near each other, which makes AI harder to distinguish which cell is safe. In addition, the result of hard mode is very low, I would say that this is also because of the density of mines.

When the number of initial safe cells are (#cells)/10:

|  |  |  |  |
| --- | --- | --- | --- |
| Difficulty | easy | medium | hard |
| Win rate (%) | 95 | 95 | 25 |

Now, I raise the number of initial safe cells. This can make our AI get more clauses in the beginning. This time AI can get a 25% win rate in the hard mode.

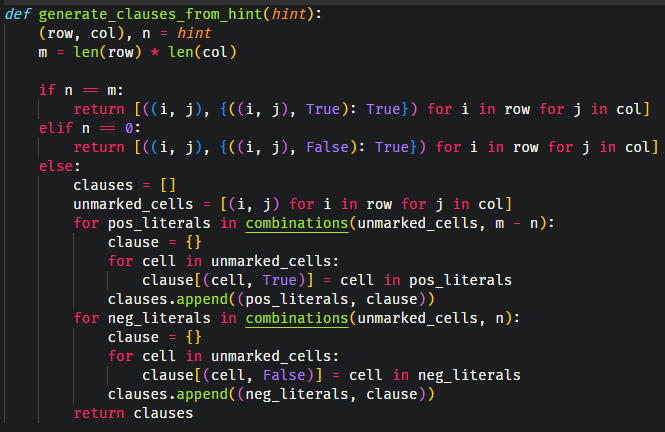
When the density of mines are (#cells)/10:

|  |  |  |  |
| --- | --- | --- | --- |
| Difficulty | easy | medium | hard |
| Win rate (%) | 91 | 94 | 93 |

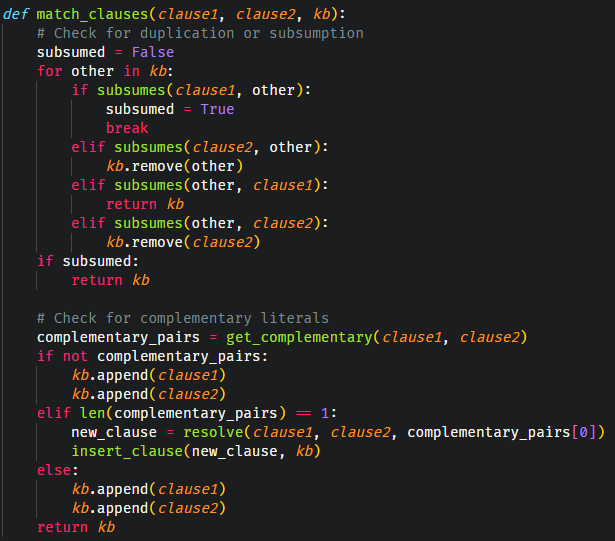
This time I change the original density, making different difficulty has same mine density. As the chart show, win rates are very close between different difficulty. Therefore, I think instead of the size of the board, the density of mines are the main factor that affect AI’s performance.

**Other function**

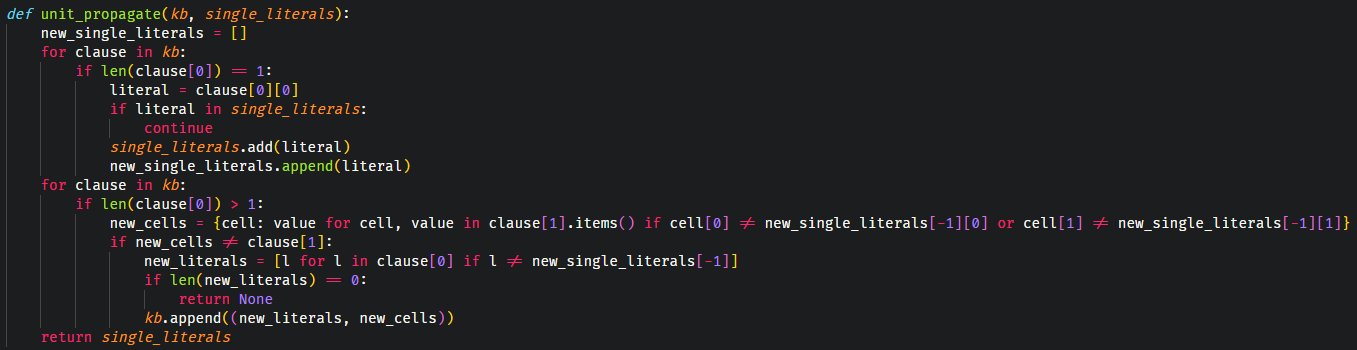
We can use hint to generate a new clause, which helps us to get new knowledge later.



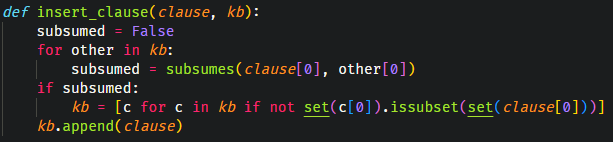
If the new clause we get is a single lateral clause, mark the cell (either safe or mine) and move that clause to KB. Next, no matter the incoming clause is single or not, we need to do matching, by which we can get some clearer clues in making safe steps. To begin with, check for duplication or subsumption. Next, check for complementary literals.



Moreover, to avoid making our KB too large, we need to implement unit-propagation. By doing this, we can ensure that the clauses we maintained in KB are all clauses that cannot be resolve with other clauses. So, if a new single-literal clause appeared, discard those multi-literal clauses containing the incoming clause if both two clauses are positive or negative., and if not, we can pull out the incoming clause from that multi-literal clause.



After we complete matching, under certain circumstances (mostly when new clause appears), we need to insert the clauses into KB. During the insertion, I first check if there the clause is identical. If not, I then start to check for subsumption.



Finally, we finish an iteration. And we can start over by making a new safe move, get new hint and clause, matching… etc.

**Appendix**

