Game control module:

- There are three mode: easy(9*9 with 10 mines), medium(16*16 with 25 mines) and hard(30*16 with 99 mines), which is saved in the property n_row, n_col and n_mines in the Game() instance game.

```
self.n_row = preset[difficulty][0]
self.n_col = preset[difficulty][1]
self.n_grid = self.n_row*self.n_col
self.n_mine = preset[difficulty][2]
self.board = [] # to record where are the mines, -1: mines, others: how many mines in surrounding grids
self.marked = [] # to record what grids are marked by the player, 0: not marked, 1: marked safe, 2: marked
for i in range(self.n_row):
    tmp = []
    tmp2 = []
    for j in range(self.n_col):
        tmp.append(0)
        tmp2.append(0)
    self.board.append(tmp)
    self.marked.append(tmp2)
cur_n_mine = 0
while cur_n_mine!=self.n_mine:
    r = rd.randint(0, self.n_row-1)
    c = rd.randint(0, self.n_col-1)
    if self.board[r][c]!=-1: # not mined yet
        self.board[r][c] = -1
        cur_n_mine += 1
```

- Define a query method to return the unmarked mines nearby the queried cell and the list of the position of the cells

```
def query(self, row: int, col: int):
    if self.board[row][col]==-1: # the queried grid has a mine
        print("This is mined! You lose!")
        return (-1, [])
    else:
        ret = list()
        unmarked_mine = self.board[row][col]
        for r in [-1, 0, 1]:
            for c in [-1, 0, 1]:
                if row+r>=0 and row+r<self.n_row and col+c>=0 and col+c<self.n_col: # inside the game board
                    if r==0 and c==0:
                        continue
                    if self.marked[row+r][col+c]==0: # unmarked
                        ret.append((row+r, col+c))
                    elif self.marked[row+r][col+c]==2: # if the marked grid is marked mined
                        unmarked_mine -= 1

        return (unmarked_mine, ret)
```

- Define a method to return safe list using random, where the size of the initial safe list is round(sqrt(#cells))

```
self.safe_list = []
while len(self.safe_list)<round(math.sqrt(self.n_grid)): # fill the safe list
    r = rd.randint(0, self.n_row-1)
    c = rd.randint(0, self.n_col-1)
    if self.board[r][c]!=-1 and (0, r, c) not in self.safe_list: # safe and not in the safe list
        self.safe_list.append((0, r, c))
```

Player module:

- Initialized KB with the safe list gotten from the game module, and KB0 with an empty list.

```python
def __init__(self, safe_list, n_row, n_col, n_mine):
    # each tuple is a lateral, tuple[0] = 0 for negative(safe), 1 for positive(mined)
    self.KB = list() # use a list to maintain KB, a list of CNF clauses (each clause is a list of tuples)
    for tuples in safe_list:
        self.KB.append(set({tuples}))
    self.KB0 = list() # use a list to maintain KB0, a list of single-lateral clauses of marked cells
    self.marked = list() # record which grids are marked
    for i in range(n_row):
        tmp = list()
        for j in range(n_col):
            tmp.append(False)
        self.marked.append(tmp)
    self.mine_left = n_mine
    self.n_unmarked = n_row*n_col
```

Game flow:

- Using propositional logic to infer where is mined and where is safe, which contains CNF (KB and KB0), clauses and laterals. I used a list to maintain KB and another to maintain KB0(lists of sets, i.e. lists of clauses), and used set to represent the clauses (sets of tuples, i.e. sets of laterals) and used three-tuple to represent the laterals. The first element indicates positive or negative, the second and the third represent the row and column of the grid respectively.
- Proceed the game with a while loop, escape when #marked cells equals to #mines (win) or encounters a stuck condition (stuck) or querying a mine (lose).

```python
def query(self, row: int, col: int):
    if self.board[row][col]==-1: # the queried grid has a mine
        print("This is mined! You lose!")
        return (-1, [])
```

- Within each iteration:
  - if there is a single-lateral clause in the KB:
    - mark it safe when it is a negative lateral, otherwise unsafe.

```python
def mark(self, row, col, status):
    if status==0: # safe
        self.marked[row][col] = 1
    else: # unsafe
        self.marked[row][col] = 2
```

    - move it to KB0 and record #unmarked mine-1

```python
game.mark(list(i)[0][1], list(i)[0][2], list(i)[0][0])
player.marked[list(i)[0][1]][list(i)[0][2]] = True
player.n_unmarked -= 1
if list(i)[0][0]==1: # mined
    player.mine_left -= 1
player.KB0.append(i)
```

- Process the matching of that clause, if the two clauses of only one complementary lateral, then eliminate them and then return the concatenation of them, otherwise return None (de nothing). Before the insertion into KB, do resolution and subsumption to reduce # of laterals in the clause and prevent against unnecessary duplication.

```python
def matching(c1: set, c2: set):
    n_comp = 0
    comp = set()
    for i in c1:
        for j in c2:
            if i[0]!=j[0] and i[1]==j[1] and i[2]==j[2]: # i and j are complementary laterals
                n_comp += 1
                if n_comp>=2: # more than one complementary laterals
                    return None
                comp.update(c1)
                comp.update(c2)
                comp.remove(i)
                comp.remove(j)

    if n_comp!=1:
        return None
    return comp
```

```python
# process the matching of that clause to all the remaining clauses in the KB
for j in player.KB:
    if i!=j:
        new_clause = matching(i, j) # return None if no or more than two complementary lateral
        if new_clause!=None:
            new_clause = resolution(new_clause, player.KB0)
            new_KB = subsumption(new_clause, new_KB)
```

- If the cell is safe, query the game control module and insert the clauses of its unmarked neighbors. Same as the above, do resolution and subsumption to prevent against unnecessary duplication.

```python
# if the cell is safe
if list(i)[0][0]==0:
    # Query the game control module for the hint at that cell
    (n_mine, unmarked_cells) = game.query(list(i)[0][1], list(i)[0][2])
    if n_mine==-1:
        return "lose"
    # insert the clauses regarding its unmarked neighbors into the KB
    if n_mine==len(unmarked_cells):
        # Insert the m single-literal positive clauses to the KB, one for each unmarked cell.
        for cell in unmarked_cells:
            new_KB.append(set({(1, cell[0], cell[1])}))
    elif n_mine==0:
        # Insert the m single-literal negative clauses to the KB, one for each unmarked cell.
        for cell in unmarked_cells:
            new_KB.append(set({(0, cell[0], cell[1])}))
    else:
        # generate CNF clauses
        # positive
        for new_clause in generate_n_clause(unmarked_cells, len(unmarked_cells)-n_mine+1, 1):
            new_KB = subsumption(new_clause, new_KB)
        # negative
        for new_clause in generate_n_clause(unmarked_cells, n_mine+1, 0):
            new_KB = subsumption(new_clause, new_KB)
```

- If it is not a single-lateral clause:

◆ Apply pairwise matching of the clauses in the KB with one of those being a two-lateral clause. This is to prevent the overgrow of the KB size. Same as the above, do resolution and subsumption to prevent against unnecessary duplication.

```
if len(i)==2: # keep KB from growing too fast
    for j in player.KB:
        if i!=j:
            new_clause = matching(i, j) # return None if no or more than two complementary lateral
            if new_clause!=None:
                new_clause = resolution(new_clause, player.KB0)
                new_KB = subsumption(new_clause, new_KB)
    new_clause = resolution(i, player.KB0)
    new_KB = subsumption(new_clause, new_KB)
new_clause = resolution(i, player.KB0)
new_KB = subsumption(new_clause, new_KB)
```

Game termination

- End the game when one of the three situations met:
  - Win: All cells are marked.

```
Puzzle solved successfully!
  0  0  1 -1  2  1  0  0  0  0  1 -1  1  1  1  1
  0  0  1  2 -1  1  0  1  1  1  1  1  1  1 -1  1
  0  1  1  2  2  2  1  1 -1  2  1  1  0  1  1  1
  1  2 -1  1  1 -1  1  1  1  2 -1  1  0  0  0  0
  2 -1  3  1  1  1  1  0  0  1  1  1  0  1  1  1
  2 -1  2  1  1  1  0  0  0  0  0  0  0  1 -1  1
  2  2  1  1 -1  1  0  0  0  0  0  0  0  1  1  1
 -1  1  1  2  2  1  0  0  1  1  1  1  1  1  0  0
  1  1  2 -1  2  0  0  0  1 -1  1  1 -1  1  0  0
  0  0  3 -1  3  0  0  0  1  1  1  1  1  1  0  0
  0  0  2 -1  2  0  0  0  1  1  1  0  0  0  0  0
  0  0  1  1  1  1  1  2  2 -1  1  0  0  0  0  0
  0  1  1  2  1  2 -1  2 -1  2  1  0  0  0  0  0
  0  1 -1  2 -1  2  2  3  2  1  0  0  0  0  1  1
  0  1  1  2  1  1  1 -1  1  0  0  0  0  0  1 -1
  0  0  0  0  0  0  1  1  1  0  0  0  0  0  1  1
```

  - Stuck: There are several choices to fill in the cells, which cannot be inferred by the resolution algorithm.

```
Game stuck!
  0  0  0  0  0  1 -1  2 -1
  0  1  1  1  0  1  1  2  1
  1  2 -1  2  1  1  0  1  1
 -1  3  2  3 -1  1  0  1 -1
  1  2 -1  2  1  1  0  1  1
  0  1  2  2  1  1  1  1  0
  1  1  2 -1  1  1 -1  1  0
  1 -1  2  1  1  1  1  1  0
  1  1  1  0  0  0  0  0  0
```

  - Lose: terminated if querying the mined cell.

About generating clauses from the hints

- When there are n mines in m unmarked cells:
  - If n==m: insert all the cells with positive lateral into the KB

```python
if n_mine==len(unmarked_cells):
    # Insert the m single-literal positive clauses to the KB, one for each unmarked cell.
    for cell in unmarked_cells:
        new_KB.append(set({(1, cell[0], cell[1])}))
```

- If n==0: insert all the cells with negative lateral into the KB

```python
# Insert the m single-literal negative clauses to the KB, one for each unmarked cell
for cell in unmarked_cells:
    new_KB.append(set({(0, cell[0], cell[1])}))
```

- Else: insert all pairs of positive(at least n+1 cells contain one mine) and negative(at least m-n+1 cells contain a safe cell) laterals.

```python
# generate CNF clauses
# positive
for new_clause in generate_n_clause(unmarked_cells, len(unmarked_cells)-n_mine+1, 1)
    new_KB = subsumption(new_clause, new_KB)
# negative
for new_clause in generate_n_clause(unmarked_cells, n_mine+1, 0):
    new_KB = subsumption(new_clause, new_KB)
```

About inserting a new clause to the KB

- Resolution: resolution with KB reduce the size of the clauses.

```python
def resolution(s: set, KB0: list):
    c = s.copy()
    for clause in KB0:
        tmp = matching(c, clause)
        if tmp!=None:
            c = tmp
    return c
```

- Subsumption: subsumption removes rules that can entail another rules.

```python
# check subsumption
def subsumption(clause: set, KB: list):
    if len(clause)==0: # if the new clause is an empty set
        return KB
    clause_is_sup = False # record whether the new clause is the superset of any member in KB
    new_KB = []
    for i in KB:
        if clause.issuperset(i):
            clause_is_sup = True
            new_KB.append(i)
        elif clause.issubset(i):
            continue
        else:
            new_KB.append(i)
    if not clause_is_sup:
        new_KB.append(clause)
    return new_KB
```

Conclusion

- I Run 100 times for each test and find that sometimes it will stuck, with the test, there are 89 successful rate in easy, 92 in medium and 62 in hard. Getting more cells from the safelist can improve the successful rate in the performance. As it retrieves more separate cells in the map, which is likely to form a useful inference rule when connecting with other parts.
- It takes a long time to run as the # of cells increases, running easy mode for 100 times takes less than a minute, medium takes about 3 minutes, while hard takes about 20 minutes to finish the game. I wonder if there are a better way to improve the performance other than subsumption and resolution.
- I use ANSI escape code to print color in the console, with unmarked cells in grey, safe in green and mined in yellow.
- I've a hard time debugging the code and finally figured out that I somehow forgot to append the original rules back to the KB when one applying resolution algorithm.

Appendix

```
import numpy as np
import random as rd
import math
import os

os.system('color')

preset = {
	"easy": (9, 9, 10),
	"medium": (16, 16, 25),
	"hard": (30, 16, 99)
}

class Game():
	def __init__(self, difficulty): # initialize the game
		self.n_row = preset[difficulty][0]
		self.n_col = preset[difficulty][1]
		self.n_grid = self.n_row*self.n_col
		self.n_mine = preset[difficulty][2]
```

```
        self.board = [] # to record where are the mines, -1: mines, others: how
many mines in surrounding grids
        self.marked = [] # to record what grids are marked by the player, 0: not
marked, 1: marked safe, 2: marked mined
        for i in range(self.n_row):
            tmp = []
            tmp2 = []
            for j in range(self.n_col):
                tmp.append(0)
                tmp2.append(0)
            self.board.append(tmp)
            self.marked.append(tmp2)
        cur_n_mine = 0
        while cur_n_mine!=self.n_mine:
            r = rd.randint(0, self.n_row-1)
            c = rd.randint(0, self.n_col-1)
            if self.board[r][c]!=-1: # not mined yet
                self.board[r][c] = -1
                cur_n_mine += 1

        self.safe_list = []
        while len(self.safe_list)<round(math.sqrt(self.n_grid)): # fill the safe list
            r = rd.randint(0, self.n_row-1)
            c = rd.randint(0, self.n_col-1)
            if self.board[r][c]!=-1 and (0, r, c) not in self.safe_list: # safe and not in
the safe list
                self.safe_list.append((0, r, c))

        for i in range(0, self.n_row):
            for j in range(0, self.n_col):
                if self.board[i][j]==-1:
                    continue
                for r in [-1, 0, 1]:
                    for c in [-1, 0, 1]:
                        if i+r>=0 and i+r<self.n_row and j+c>=0 and
j+c<self.n_col: # inside the game board
                            if self.board[i+r][j+c]==-1: # its surrounding grid is
mined
```

```
                                self.board[i][j] += 1

    # return the number of mines and the unmarked cells surrounding the queried one. A list of two tuples(row, col)
    def query(self, row: int, col: int):
        if self.board[row][col]==-1: # the queried grid has a mine
            print("This is mined! You lose!")
            return (-1, [])
        else:
            ret = list()
            unmarked_mine = self.board[row][col]
            for r in [-1, 0, 1]:
                for c in [-1, 0, 1]:
                    if row+r>=0 and row+r<self.n_row and col+c>=0 and col+c<self.n_col: # inside the game board
                        if r==0 and c==0:
                            continue
                        if self.marked[row+r][col+c]==0: # unmarked
                            ret.append((row+r, col+c))
                        elif self.marked[row+r][col+c]==2: # if the marked grid is marked mined
                            unmarked_mine -= 1

            return (unmarked_mine, ret)

    # return the safe_list. A list of three tuples(safe, row, col)
    def get_safe_list(self):
        return self.safe_list

    def print_board(self):
        for i in range(self.n_row):
            for j in range(self.n_col):
                if self.marked[i][j]==0: # not marked
                    print("\033[90m {:2} \033[00m".format(self.board[i][j]), end="")
                elif self.marked[i][j]==1: # marked safe
                    print("\033[92m {:2} \033[00m".format(self.board[i][j]), end="")
```

```
                else:
                    print("\033[93m {:2} \033[00m".format(self.board[i][j]),
end="")
            print("")

    def mark(self, row, col, status):
        if status==0: # safe
            self.marked[row][col] = 1
        else: # unsafe
            self.marked[row][col] = 2

class Player():
    def __init__(self, safe_list, n_row, n_col, n_mine):
        # each tuple is a lateral, tuple[0] = 0 for negative(safe), 1 for
positive(mined)
        self.KB = list() # use a list to maintain KB, a list of CNF clauses (each clause is
a list of tuples)
        for tuples in safe_list:
            self.KB.append(set({tuples}))
        self.KB0 = list() # use a list to maintain KB0, a list of single-lateral clauses of
marked cells
        self.marked = list() # record which grids are marked
        for i in range(n_row):
            tmp = list()
            for j in range(n_col):
                tmp.append(False)
            self.marked.append(tmp)
        self.mine_left = n_mine
        self.n_unmarked = n_row*n_col

# return the matching of the two sets when there's exactly one complementary
lateral
def matching(c1: set, c2: set):
    n_comp = 0
    comp = set()
    for i in c1:
        for j in c2:
            if i[0]!=j[0] and i[1]==j[1] and i[2]==j[2]: # i and j are complementary
```

```
laterals
                                n_comp += 1
                                if n_comp>=2: # more than one complementary laterals
                                        return None
                                comp.update(c1)
                                comp.update(c2)
                                comp.remove(i)
                                comp.remove(j)

        if n_comp!=1:
                return None
        return comp

# return the resulting clause after resolution with KB0
def resolution(s: set, KB0: list):
        c = s.copy()
        for clause in KB0:
                tmp = matching(c, clause)
                if tmp!=None:
                        c = tmp
        return c

# l is the clause used to generate new clauses
# n is the size of each new clause
# safe indicates positive or negative literals
# return a list of clauses(set)
def generate_n_clause(l: list, n: int, safe: int):
        ret = []
        x = len(l)
        for i in range(1 << x):
                tmp = set()
                for j in range(x):
                        if i & (1 << j):
                                tmp.add((safe, l[j][0], l[j][1]))
                if len(tmp)==n:
                        ret.append(tmp)
        return ret
```

```
# check subsumption
def subsumption(clause: set, KB: list):
        if len(clause)==0: # if the new clause is an empty set
                return KB
        clause_is_sup = False # record whether the new clause is the superset of any
member in KB
        new_KB = []
        for i in KB:
                if clause.issuperset(i):
                        clause_is_sup = True
                        new_KB.append(i)
                elif clause.issubset(i):
                        continue
                else:
                        new_KB.append(i)
        if not clause_is_sup:
                new_KB.append(clause)
        return new_KB

def play_game(game: Game, player: Player):
        new_rule = 0
        global_rule_add = False
        last_KB0_length = -1
        while True: # game is not finished
                if len(player.KB0)==game.n_grid: # All grids are marked
                        break
                new_KB = []
                for i in player.KB:
                        if len(i)==1: # single lateral clause
                                if i not in player.KB0:
                                        game.mark(list(i)[0][1], list(i)[0][2], list(i)[0][0])
                                        player.marked[list(i)[0][1]][list(i)[0][2]] = True
                                        player.n_unmarked -= 1
                                        if list(i)[0][0]==1: # mined
                                                player.mine_left -= 1
                                        player.KB0.append(i)
                                        # process the matching of that clause to all the remaining
clauses in the KB
```

```
                    for j in player.KB:
                        if i!=j:
                            new_clause = matching(i, j) # return None if no or
more than two complementary laterals, new set if exactly one
                            if new_clause!=None:
                                new_clause = resolution(new_clause,
player.KB0)
                                new_KB = subsumption(new_clause,
new_KB)
                    # if the cell is safe
                    if list(i)[0][0]==0:
                        # Query the game control module for the hint at that
cell
                        (n_mine, unmarked_cells) = game.query(list(i)[0][1],
list(i)[0][2])
                        if n_mine==-1:
                            return "lose"
                        # insert the clauses regarding its unmarked neighbors
into the KB
                        if n_mine==len(unmarked_cells):
                            # Insert the m single-literal positive clauses to the
KB, one for each unmarked cell.
                            for cell in unmarked_cells:
                                new_KB.append(set({(1, cell[0], cell[1])}))
                        elif n_mine==0:
                            # Insert the m single-literal negative clauses to the
KB, one for each unmarked cell.
                            for cell in unmarked_cells:
                                new_KB.append(set({(0, cell[0], cell[1])}))
                        else:
                            # generate CNF clauses
                            # positive
                            for new_clause in
generate_n_clause(unmarked_cells, len(unmarked_cells)-n_mine+1, 1):
                                new_KB = subsumption(new_clause,
new_KB)
                            # negative
                            for new_clause in
```

```
generate_n_clause(unmarked_cells, n_mine+1, 0):
                                new_KB = subsumption(new_clause,
new_KB)
            else:
                # Apply pairwise matching of the clauses in the KB
                # process the matching of that clause to all the remaining clauses
in the KB
                new_rule += 1
                if len(i)==2: # keep KB from growing too fast
                    for j in player.KB:
                        if i!=j:
                            new_clause = matching(i, j) # return None if no or
more than two complementary laterals, new set if exactly one
                            if new_clause!=None:
                                new_clause = resolution(new_clause,
player.KB0)
                                new_KB = subsumption(new_clause,
new_KB)
                    new_clause = resolution(i, player.KB0)
                    new_KB = subsumption(new_clause, new_KB)
                new_clause = resolution(i, player.KB0)
                new_KB = subsumption(new_clause, new_KB)

        # do propagation
        propagation_KB = new_KB
        for clause in new_KB:
            clause = resolution(clause, player.KB0)
            propagation_KB = subsumption(clause, propagation_KB)

        if last_KB0_length==len(player.KB0): # no new rules inferred
            if player.n_unmarked<=15 and not global_rule_add:
                global_rule_add = True
                # Add a rule regarding how many mines left in the remaining
unmarked areas
                new_clause = set()
                unmarked = list()
                for i in range(len(player.marked)):
                    for j in range(len(player.marked[0])):
```

```
                    if player.marked[i][j]==False: # not marked yet
                        unmarked.append((i, j))

                # generate CNF clauses
                # positive
                for new_clause in generate_n_clause(unmarked, len(unmarked)-
player.mine_left+1, 1):
                    new_clause = resolution(new_clause, player.KB0)
                    new_KB = subsumption(new_clause, new_KB)
                # negative
                for new_clause in generate_n_clause(unmarked,
player.mine_left+1, 0):
                    new_clause = resolution(new_clause, player.KB0)
                    new_KB = subsumption(new_clause, new_KB)
                player.KB = new_KB
                continue
            if new_rule<=5: # add clauses of 2 laterals into resolution
                continue
            else: # game is probably stuck
                print("Game stuck!")
                return "stuck"

        player.KB = propagation_KB
        last_KB0_length = len(player.KB0)
        new_rule = 0
    print("Puzzle solved successfully!")
    game.print_board()
    return "win"

record = dict({
    "win": 0,
    "lose": 0,
    "stuck": 0
})
for i in range(100):
    game = Game("medium")
    player = Player(game.get_safe_list(), game.n_row, game.n_col, game.n_mine)
    record[play_game(game, player)] += 1
```

```
print(record)
```