資工3B 0616026 游騰德

Introduction to AI

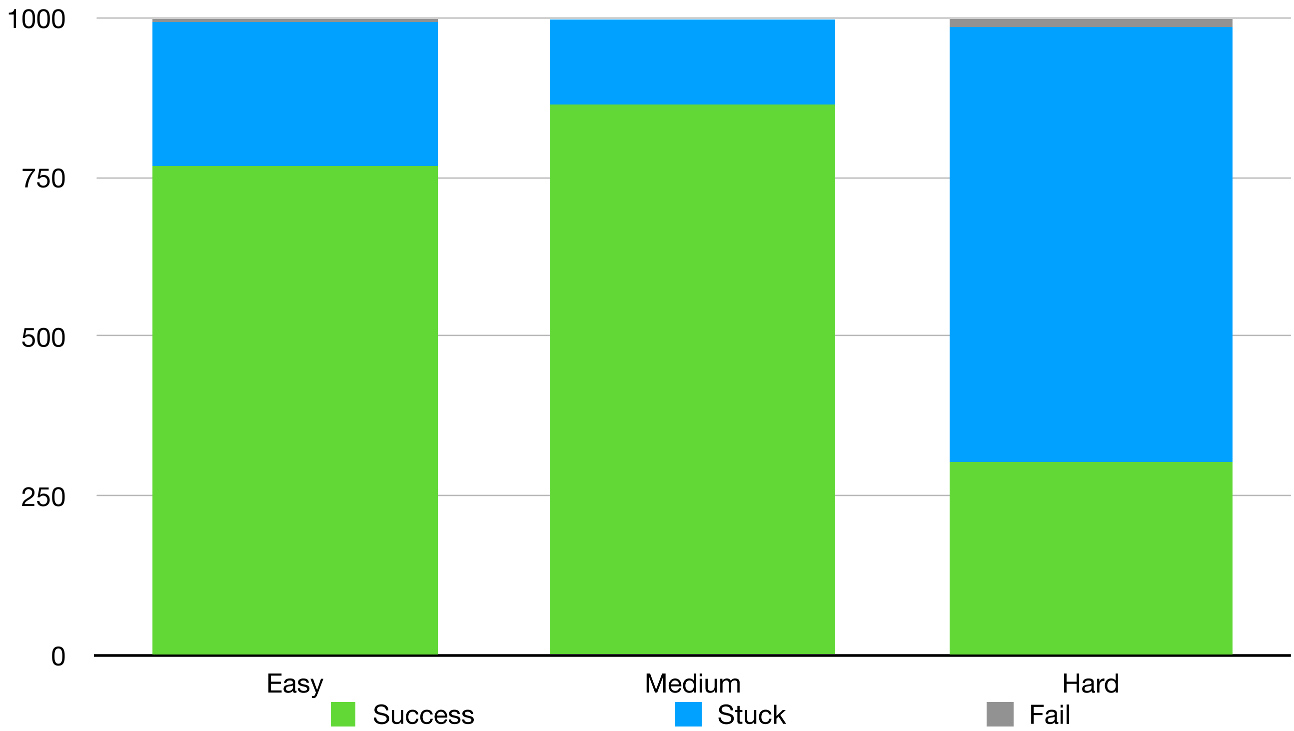
Programming Assignment #3

實驗與結果

盤面難易度與結果比較

表 1 為使用 3 種不同難易程度（ Easy 、 Medium 、 Hard ）的盤面，測試每種盤面難易度時，均隨機產生 1000 組盤面進行測試，進行遊戲後統計 Success 、 Stuck 、 Fail 的結果紀錄，表 1 Y 軸為場數，使用線性刻度。

表 1



遊戲開始時，均給定 round(sqrt(#cells)) 個初始安全位置， Easy 盤面大小為 9*9 ，有 10 個地雷；Medium 盤面大小為 16*16 ，有 25 個地雷；Hard 盤面大小為 30*16 ，有 75 個地雷（因使用 Spec 的要求而放置 99 個地雷時，遊戲實在太容易 Stuck ，故改為 75 個地雷進行測試） 。

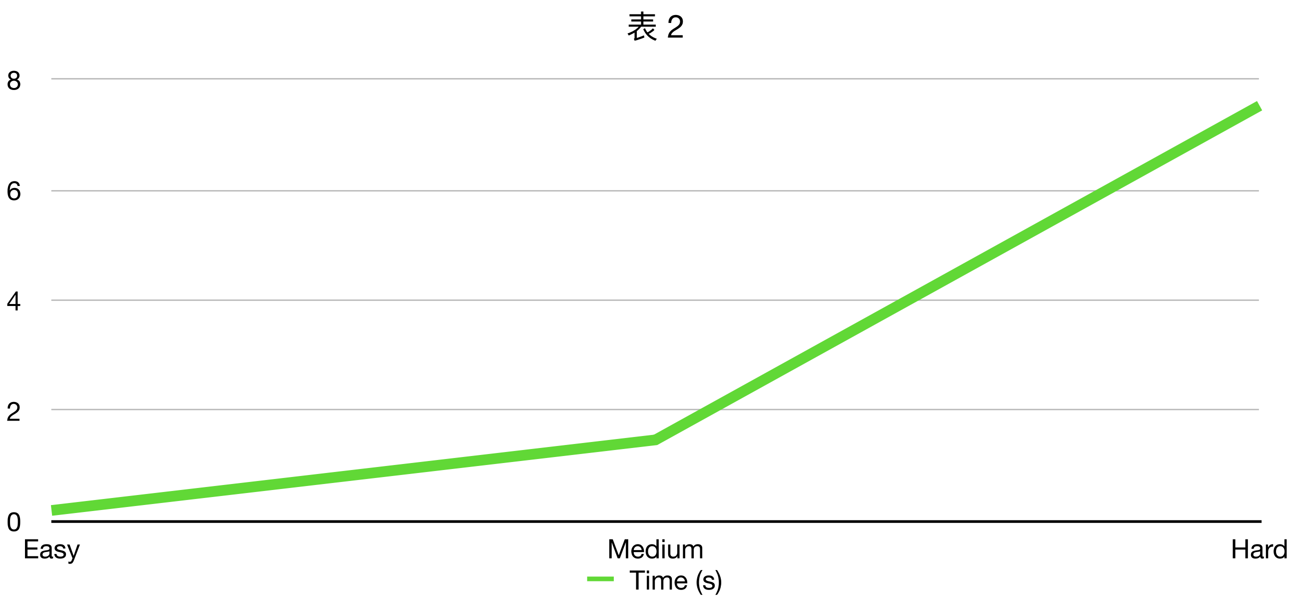
從表 1 可以看出，遊戲在難易程度為 Medium 時，最容易成功解出，其 Success 比率約為 0.86 ，在難易程度為 Easy 時，其 Success 比率約為 0.76 ，難易程度為 Hard 時，最容易 Stuck ，其 Success 比率僅約為 0.3 。

表 1 實驗中，若遞迴 3 次仍無法產生新的步數，即判定為 Stuck ，另外，其所發生的極少數 Fail ，大都是因一些並排的地雷相鄰圍繞行成一個封閉區域，使得遊戲時無法探索封閉區域內的 Hints ，也無法對封閉區域中的位置產生相關的 Clause ，導致即便 KB 內已無任何 Clause 時，遊戲仍尚未完成。

以下其他實驗若無特別說明，則其盤面大小及地雷數量皆與表 1 中的實驗配置相同。

盤面難易度與時間比較

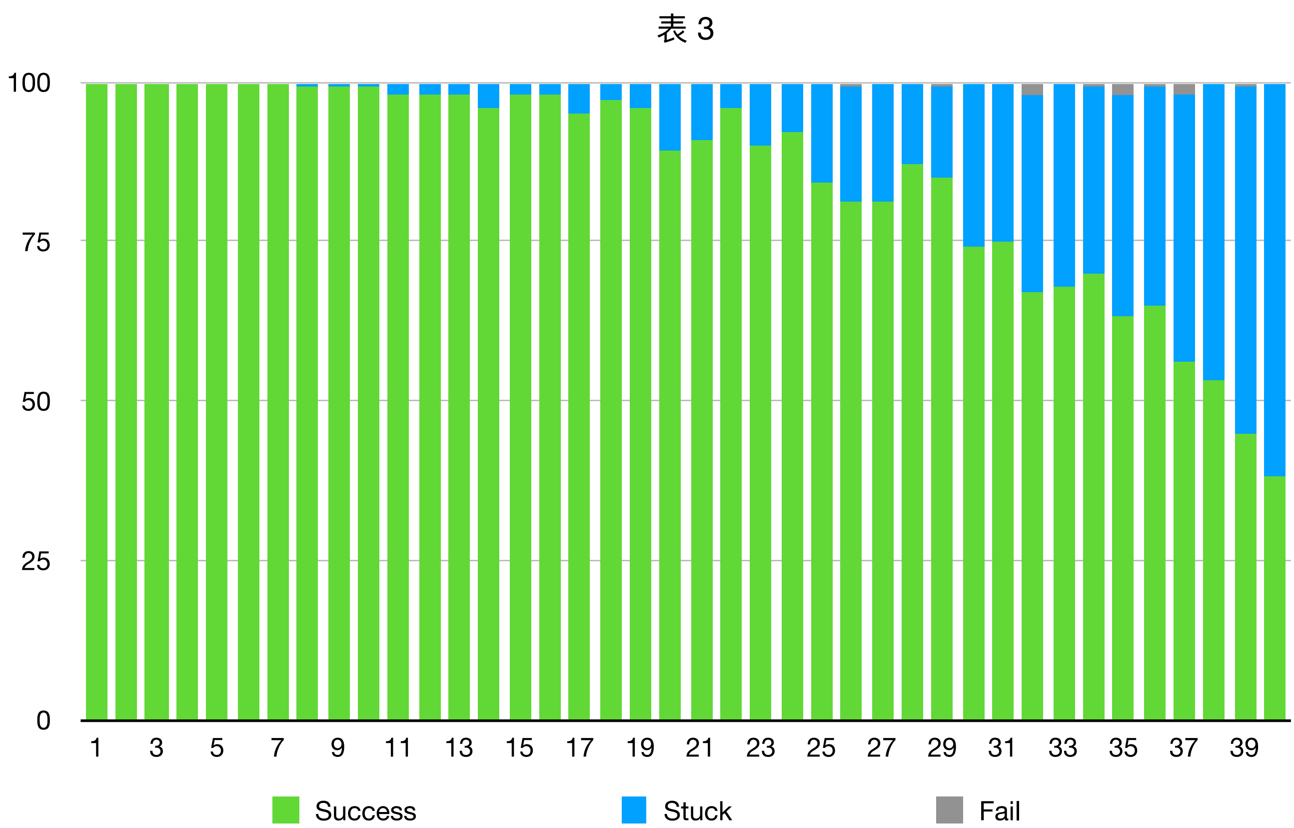
表 2 為在表 1 實驗中，成功解出盤面後，統計其時間，取其結果之平均時間紀錄。表 2 Y 軸為時間，使用線性刻度，單位為秒。



由表 2 可以看出，盤面難易度增加時，所需時間亦增加。

地雷數量與結果比較

表 3 為使用 16*16 的盤面大小，分別測試在盤面上放置 1 個至 40 個地雷，觀察其解出所需時間。測試每個地雷數量時，均隨機產生 100 組盤面進行測試，進行遊戲後統計 Success 、 Stuck 、 Fail 的結果紀錄，表 3 X 軸為地雷數量， Y 軸為場數，使用線性刻度。

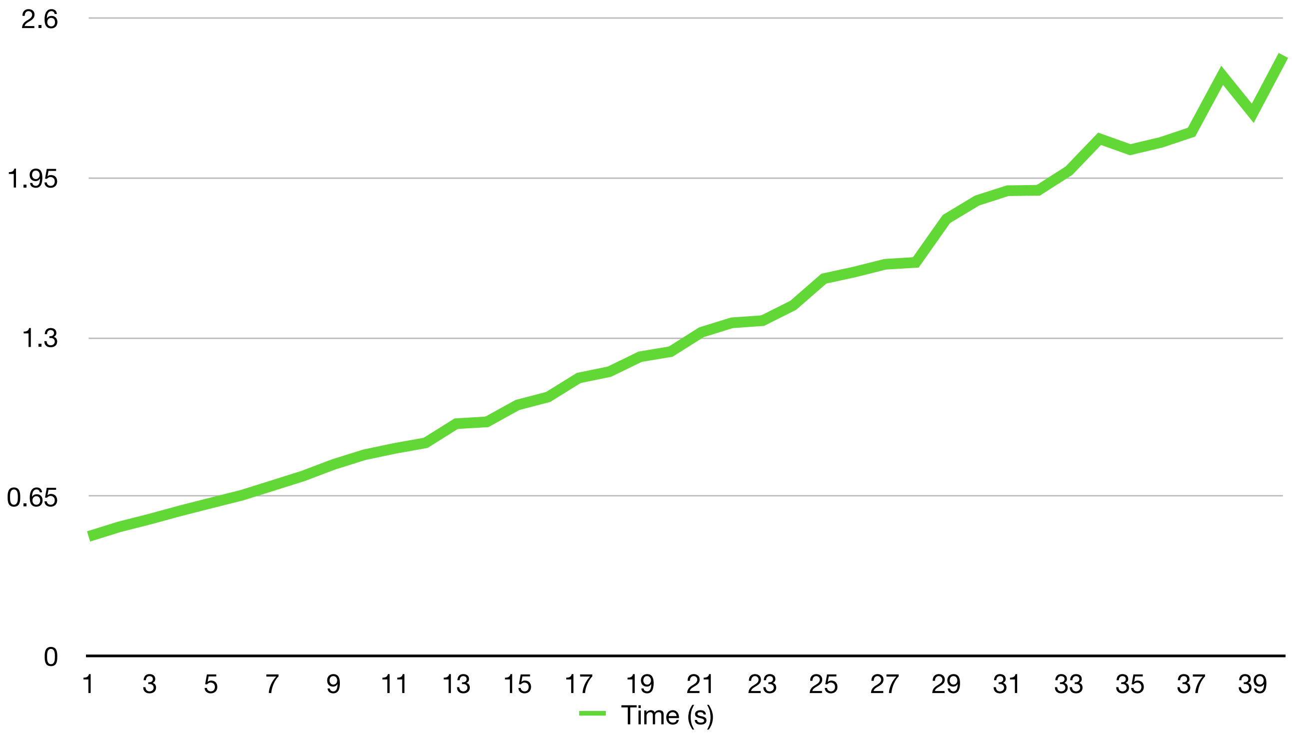


在表 3 實驗中，最初均給定 round(sqrt(#cells)) 個初始安全位置，從表 3 可以看出，當地雷數量增加時，遊戲越來越容易發生 Stuck 的情形，我認為這是由於在確定是地雷的位置上，無法得知該位置的 Hints 值，也就是說，當地雷數量越多，所能取得的 Hints 數量就越少，導致越來越容易 Stuck 。

地雷數量與時間比較

表 4 為在表 3 實驗中，成功解出盤面後，統計其時間，取其結果之平均時間紀錄。表 3 X 軸為地雷數量， Y 軸為時間，使用線性刻度，單位為秒。

表 4



從表 4 可以看出，即便是盤面大小皆相同，當地雷數量越來越多時，所需時間仍成線性成長。我認為這是由於地雷數量越多，所能取得的 Hints 數量就越少，使得需要做更多的 Pairwise matching 才能產生 Single literal 的原因所導致。

所學

透過本次實驗我所學以下幾點：

1. 利用物件導向程式設計，以 Python 實作互動遊戲之 Framework。
2. 了解 Propositional logic 、 Resolution 、 Matching 等等技巧的運作原理、特性以及優缺點。
3. 透過比較不同的盤面難易程度來體會時間及空間複雜度對於 Propositional logic 的影響。
4. 將實驗結果整理、列表並使人易於閱讀。

討論

使用 First order logic

定義 Relations ： hint_eq_around_mines(x) 表示 x 的 Hints 和周圍已知的地雷數量相同、adjacent(x, y) 表示 x 、 y 相鄰、mine(x), safe(x), unmarked(x)表示 x 的狀態。

1. 對於所有安全的位置 x ，若其 Hints 和周圍已知的地雷數量相同，則與 x 相鄰的所有未知位置為安全。
   ∀x ∀y safe(x) ∧ hint_eq_around_mines(x) ∧ adjacent(x, y) ∧ unmarked(y) ⇒ safe(y)

2. 對於所有安全的位置 x ，若其 Hints 和周圍已知的地雷數量不相同，則與 x 相鄰的未知位置中存在地雷。
   ∀x ∃y safe(x) ∧ ¬hint_eq_around_mines(x) ∧ adjacent(x, y) ∧ unmarked(y) ⇒ mine(y)

以上為兩個使用 First order logic 的 Sentence 實例，若 Sentence 完整，我推測使用 First order logic 會比使用 Propositional logic 更有效率。

Forward chaining 與 Backward chaining

使用 Forward chaining 時，先將 KB 做成圖，其中每個節點都是一個 Implication ，遞迴計算每個節點當中已被證實的 Fact ，當布標節點被標示為 1 時，即代表目標亦成為 Fact 了，整個過程即為 Horn clause implication 的拆除。

使用從目標節點出發，遞迴檢視可以 Imply 目標的 Horn clause，直到沒有 Horn clause 之後，再檢查實際是否能推到目標。

Backward chaining 中的 Goal-driven 特性使得其與 Forward chaining 相比更有效率。

Stuck 時的猜測

如果剩餘未解位置不多，可以嘗試加入 Global constraint 進入 KB ，使用地雷數量與剩餘為標記的位置來增加線索。

若無法使用 Global constraint 或使用後無效，可使用表決機制，即用 KB 中的 Clause 裡每個 Literal 來投票，假如一個 Literal 指出位置 A 為安全，則 A 加 1 票，若不安全則減 1 票，最後統計票高者進行猜測並選擇。

使用 Search

我認為邏輯和搜尋可以同時並進，也就是每次遞回時先使用邏輯標示出已經確定的位置，然後避開這些位置進行搜尋。如此一來，搜尋時會減少搜索不必要的、已被推定為安全或是地雷的位置，可以減少搜尋所需的時間和空間，同時也可避免在遊戲初始時，無意義的產生很多 Clause 進行無用的 Matching。

疑問與探討

1. 在盤面難易度為 Hard 時， Spec 所要求的地雷數量為 99 ，但當我使用這項配置時，遊戲非常容易進入 Stuck 狀態，雖然不排除是我的程式有錯誤，但我有將盤面狀態印出並逐一比對，也確認真的是 Stuck ，因此很疑惑為何會有這樣的狀況發生。此外，即便是將原本 99 個地雷的配置修改為 75個地雷，成功率依然只有如表 1 所標示的三成左右，遠遠不如難易度 Medium 近九成的成功率。
2. 在 Spec 中有提到可以將 round(sqrt(#cells)) 個初始安全位置的數量加以改變，觀察遊戲成功機率的變化。而我認為：其實初始位置最重要的是要有 0 的 Hints ，如果初始位置完全沒有 Hints 為 0 ，則遊戲有極高的可能性會一開局就進入 Stuck 裝態。然而，round(sqrt(#cells)) 因涉及根號，因此不好改變與掌握，所以我稍作變形，改做了表 3 和表 4 的實驗，我認為其原理相當，都是在於探討為 0 的 Hints 在初始安全位置中出現的機率，機率越大，越不容易 Stuck，反之就越容易。

未來發想

在實驗的過程中，如果印出 KB 來做觀察，會發現單純使用 Propositional logic 時，其所產生的很多 Clause 其實是無用的，但是這個 Clause 會一直存在 KB 當中，不斷的與其他 Clause 進行毫無用處的 Matching ，最後才被移除。而這樣的狀況往往是因為有一個不為 0 的 Hints ，其位置的周圍完全沒有其他 Hints 或已被標示為地雷的位置，則此 Hints 在這樣的情形之下，對於整個遊戲來說是毫無用處的。因此，為來可考慮先暫時排除將這樣的 Hints 所產生的 Clause 列入 KB ，直到其周圍有位置被解出，如此一來可以免去很多不必要的 Matching ，我認為或許可以增加遊戲的效率。

Appendix

Structure

- minesweeper.py
- agent.py
- logic.py
- board.py
- test.py

minesweeper.py

```
import time
from board import Board, Action
from logic import Literal, Clause
from agent import Agent


class Result():
    def __init__(self, status, stuck, play_time):
        self.status = status
        self.stuck = stuck
        self.play_time = play_time


class MineSweeper():
    def __init__(self, difficulty, mines = None):
        self.board = Board(difficulty, mines)
        self.agent = Agent()
        self.debug = False

    def play(self, debug = None):
        if debug is not None:
            self.debug = debug
        start_time = time.time()
        stucked = False
        status = ''

        # Initial game
        for pos in self.board.init_safe_pos:
            new_clause = Clause( [-Literal(pos)] )
            self.agent.add_clause_to_KB(new_clause)

        while True:
            # Print current board
            if self.debug:
                self.board.print_current_board()
                print()

            # Take action
            action = self.agent.take_action(self.board)
```

```
            # Game flow
            if 'query' == action.action:
                new_hint = self.board.query(action.position)
                if new_hint == -3:
                    if self.debug:
                        print('Fail on', action.position)
                        break
                self.agent.new_hint(action.position, new_hint, 
self.board)
            elif 'mark_mine' == action.action:
                self.board.mark_mine(action.position)
            elif 'done' == action.action:
                break
            elif 'give_up' == action.action:
                stucked = True
                if self.debug:
                    print(self.agent.KB0)
                    for c in self.agent.KB:
                        print(c)
                    print('Stucked')
                break


        # Statistics
        play_time = (time.time() - start_time)
        if self.board.check_success():
            status = 'Success'
        else:
            status = 'Fail'

        if self.debug:
            # Results
            print('====Result====')
            self.board.print_current_board()
            print()
            print('====Answer====')
            self.board.print_answer_board()
            print('====Status====')
            print(status)
            print('Duration:', play_time, 'sec')

        return Result(status, stucked, play_time)

if __name__ == '__main__':
    # Examples
    game = MineSweeper('easy')
    game.play(debug = True)
```

agent.py

```
import time
import copy, itertools
from board import Action, Board
from logic import Literal, Clause

class Agent():
    def __init__(self):
        self.KB0 = []
        self.KB = []

    def add_clause_to_KB(self, new_clause):
        if not new_clause.is_empty():
            # Do resolution of the new clause with all the clauses
in KB0
            for c in self.KB0:
                tmp_clause, co_literal_count =
self.resolution(new_clause, c)
                if tmp_clause > new_clause:
                    new_clause = tmp_clause

            # Check for identication and subsumption with all the
clauses in KB
            is_useable = not (new_clause.is_empty())
            reduntents = []
            for c in self.KB:
                if not is_useable:
                    break
                if new_clause <= c:
                    is_useable = False
                if new_clause > c:
                    reduntents.append(c)
            if is_useable:
                self.KB.append(new_clause)

            for r in reduntents:
                self.remove_clause_from_KB(r)

    def remove_clause_from_KB(self, clause):
        try:
            self.KB.remove(clause)
        except:
            pass

    def global_constraint(self, b):
        unmarked, marked_mine = b.global_constraint_check()
        self.gen_clause(unmarked, marked_mine, b.mines)

    def new_hint(self, position, hint, b):
        around_unmarked = b.around_unmarked_position(position)
```

```
        around_marked_mine =
b.around_marked_mine_position(position)
        self.gen_clause(around_unmarked, around_marked_mine, hint)

    def gen_clause(self, unmarked, marked_mine, hint):
        m = len(unmarked)
        n = hint - len(marked_mine)

        if n == m:
            # Insert the m single-literal positive clauses to KB
            for au in unmarked:
                new_clause = Clause( [Literal(au)] )
                self.add_clause_to_KB(new_clause)
        elif n == 0:
            # Insert the m single-literal negative clauses to KB
            for au in unmarked:
                new_clause = Clause( [-Literal(au)] )
                self.add_clause_to_KB(new_clause)
        elif m > n > 0:
            # General cases
            # Generate CNF clauses and add them to the KB
            all_positive = list(itertools.combinations(unmarked,
m-n+1))
            all_negative = list(itertools.combinations(unmarked,
n+1))

            for comb in all_positive:
                literals = []
                for au in comb:
                    literals.append(Literal(au))
                new_clause = Clause(literals)
                self.add_clause_to_KB(new_clause)

            for comb in all_negative:
                literals = []
                for au in comb:
                    literals.append(-Literal(au))
                new_clause = Clause(literals)
                self.add_clause_to_KB(new_clause)

        else:
            # For debugging
            # print('Something is wrong')
            # print(position, hint, m, n, len(unmarked),
len(marked_mine))
            pass

    def resolution(self, clause_a, clause_b):
        co_literal_count = 0
        new_clause = Clause([])
        for l in clause_a.literals:
            literal = l
```

```
                new_clause.literals.append(literal)
        for l in clause_b.literals:
            literal = l
            co_literal = -l
            if co_literal in new_clause.literals:
                new_clause.literals.remove(co_literal)
                co_literal_count += 1
            elif literal not in new_clause.literals:
                new_clause.literals.append(literal)
        return new_clause, co_literal_count

    def remain_literals_matching(self, moved_clause,
remain_clause):
        reduntent_clause = None
        add_clause = None

        literal = moved_clause.literals[0]
        co_literal = -moved_clause.literals[0]

        if co_literal in remain_clause.literals:
            new_clause = copy.deepcopy(remain_clause)
            reduntent_clause = remain_clause
            new_clause.literals.remove(co_literal)
            add_clause = new_clause
        elif literal in remain_clause.literals:
            reduntent_clause = remain_clause

        return reduntent_clause, add_clause

    def pairwise_matching(self, clause_a, clause_b):
        # Check for duplication or subsumption first
        # Keep only the more strict clause.
        if len(clause_a.literals) > 2 and len(clause_b.literals) >
2:
            return
        if not (clause_a in self.KB and clause_b in self.KB):
            return

        if clause_a <= clause_b:
            self.remove_clause_from_KB(clause_a)
            return
        elif clause_a >= clause_b:
            self.remove_clause_from_KB(clause_b)
            return

        # Do resolution
        new_clause, co_literal_count = self.resolution(clause_a,
clause_b)

        if co_literal_count == 1:
            # Only one pair of complementary literals:
            self.remove_clause_from_KB(clause_a)
```

```
                self.remove_clause_from_KB(clause_b)
                self.add_clause_to_KB(new_clause)
            else:
                # No or more than one pairs of complementary literals
                # Do nothing
                pass

    def take_action(self, b):
        # Make a query
        no_action_count = 0

        while len(self.KB):
            if no_action_count == 3:
                return Action('give_up')
            has_single_literal = False

            for c in self.KB:
                if c.is_single_literal():
                    # Single-lateral clause in the KB
                    # Mark this cell as safe or mined
                    clause = c

                    # Move that clause to KB0
                    self.remove_clause_from_KB(clause)
                    self.KB0.append(clause)

                    # Process the matching of that clause to all
the remaining clauses in the KB
                    reduntents = []
                    adds = []
                    for remain_c in self.KB:
                        reduntent_clause, add_clause =
self.remain_literals_matching(clause, remain_c)
                        if reduntent_clause:
                            reduntents.append(reduntent_clause)
                        if add_clause:
                            adds.append(add_clause)
                    for r in reduntents:
                        self.remove_clause_from_KB(r)
                    for a in adds:
                        self.add_clause_to_KB(a)


                    if clause.is_safe():
                        return Action('query',
clause.literals[0].position)
                        no_action_count = 0
                    else:
                        return Action('mark_mine',
clause.literals[0].position)
                        no_action_count = 0
```

```
                    has_single_literal = True

            if not has_single_literal:
                no_action_count += 1
                # tmp_KB = list(self.KB)

                # Apply pairwise matching of the clauses in the KB
                # Only match clause pairs where one clause has
only at most two literals
                for comb in list(itertools.combinations(self.KB,
2)):
                    if comb[0] in self.KB and comb[1] in self.KB:
                        self.pairwise_matching(comb[0], comb[1])

                # if tmp_KB == self.KB:
                #     # self.global_constraint(b)
                #     if tmp_KB == self.KB:
                #         return Action('give_up')


            # if len(self.KB) == 0:
            #     unmarked, marked_mine = global_constraint_check
            #     if len(unmarked):
            #         self.global_constraint(b)

        return Action('done')
```

logic.py

```
class Literal():
    def __init__(self, position, positive = True):
        self.positive = positive
        self.position = position

    def __neg__(self):
        return Literal(self.position, not self.positive)

    def __eq__(self, other):
        return self.positive == other.positive and self.position
== other.position

    def __ne__(self, other):
        return not self.__eq__(other)

    def __repr__(self):
        return ('' if self.positive else '-') + str(self.position)

class Clause():
    def __init__(self, literals):
        self.literals = literals
```

```
def __eq__(self, other):
    if not len(self.literals) == len(other.literals):
        return False
    for l in self.literals:
        if l not in other.literals:
            return False
    return True

def __gt__(self, other):
    # Return True when self is stricter than other
    if not len(self.literals) < len(other.literals):
        return False
    for l in self.literals:
        if l not in other.literals:
            return False
    return True

def __ge__(self, other):
    if not len(self.literals) <= len(other.literals):
        return False
    for l in self.literals:
        if l not in other.literals:
            return False
    return True

def __lt__(self, other):
    if not len(other.literals) < len(self.literals):
        return False
    for l in other.literals:
        if l not in self.literals:
            return False
    return True

def __le__(self, other):
    if not len(other.literals) <= len(self.literals):
        return False
    for l in other.literals:
        if l not in self.literals:
            return False
    return True

def __repr__(self):
    ret = ''
    for l in self.literals:
        ret += (' v ' + str(l)) if ret else str(l)
    return '(' + ret + ')'

def is_empty(self):
    return len(self.literals) == 0

def is_single_literal(self):
    if len(self.literals) == 1:
```

```
            return True
        return False

    def is_safe(self):
        if len(self.literals) == 1:
            return not self.literals[0].positive
        return False


if __name__ == '__main__':
    # Examples
    a = Literal((5, 3))
    b = -Literal((2, 4))
    c = Literal((6, 13))
    d = -Literal((2, 1))
    e = Literal((7, 0))

    clause1 = Clause([a, b, c])
    clause2 = Clause([a, b, c, d])
    clause3 = Clause([a, b, c, d, e])
    clause4 = Clause([a, b, c, d, e])

    print(clause1 < clause2)
    print(clause3 < clause1)
    print(clause1 < clause2)
    print(clause1 < clause4)
    print(clause1 > clause4)
```

board.py

```
import copy, math, random


class Action:
    def __init__(self, action, position = None):
        self.action = action
        self.position = position


class Board():
    def __init__(self, difficulty, mines = None):
        init_param = {}
        if 'easy' == difficulty:
            init_param = {'size': (9, 9), 'mines': mines if mines
else 10}
        elif 'medium' == difficulty:
            init_param = {'size': (16, 16), 'mines': mines if
mines else 25}
        elif 'hard' == difficulty:
            init_param = {'size': (30, 16), 'mines': mines if
mines else 75}
```

```
        self.x = init_param['size'][0]
        self.y = init_param['size'][1]
        self.mines = init_param['mines']
        self.hints = []
        self.marked = []

        # Randomly generate a new board
        positions = []
        for j in range(self.y):
            for i in range(self.x):
                positions.append((i, j))

        # Select mine and initial safe positions
        init_safe_cells = round(math.sqrt(self.x * self.y))
        sltd_pos = random.sample(positions, self.mines +
init_safe_cells)
        mine_pos = sltd_pos[0:self.mines]
        self.init_safe_pos = sltd_pos[self.mines:]
        # print(mine_pos)
        # print(self.init_safe_pos)

        # Generate hints
        for i in range(self.x):
            for j in range(self.y):
                if j == 0:
                    self.hints.append([])
                    self.marked.append([])
                if (i, j) in mine_pos:
                    self.hints[i].append(-3)
                else:
                    around = self.around_position((i, j))
                    mines_count = 0
                    for a in around:
                        if a in mine_pos:
                            mines_count += 1
                    self.hints[i].append(mines_count)
                self.marked[i].append(0)

    def query(self, position):
        x = position[0]
        y = position[1]
        self.marked[x][y] = -2
        return self.hints[x][y]

    def mark_mine(self, position):
        x = position[0]
        y = position[1]
        self.marked[x][y] = -3

    def available_position(self, position):
```

```
        # Returns true if the given position is available on this 
board
        return 0 <= position[0] < self.x and 0 <= position[1] < 
self.y

    def around_position(self, position):
        # Returns a list of available postions around the given 
position
        x = position[0]
        y = position[1]
        psb_pos = [(x-1, y-1), (x, y-1), (x+1, y-1),
                   (x-1, y),              (x+1, y),
                   (x-1, y+1), (x, y+1), (x+1, y+1)]
        around = []
        for pos in psb_pos:
            if self.available_position(pos):
                around.append(pos)
        return around

    def around_unmarked_position(self, position):
        # Returns a list of unmarked postions around the given 
position
        around = self.around_position(position)
        around_unmarked = []
        for a in around:
            if self.marked[a[0]][a[1]] == 0:
                around_unmarked.append(a)
        return around_unmarked

    def around_marked_mine_position(self, position):
        # Returns a list of marked mine postions around the given 
position
        around = self.around_position(position)
        around_marked_mine = []
        for a in around:
            if self.marked[a[0]][a[1]] == -3:
                around_marked_mine.append(a)
        return around_marked_mine

    def check_success(self):
        marked_count = 0
        marked_mine_count = 0
        current = copy.deepcopy(self.marked)
        for j in range(self.y):
            for i in range(self.x):
                if current[i][j] != 0:
                    marked_count += 1
                if current[i][j] == -3:
                    marked_mine_count += 1
        return marked_count == self.x*self.y and marked_mine_count 
== self.mines
```

```
    def global_constraint_check(self):
        unmarked = []
        marked_mine = []
        current = copy.deepcopy(self.marked)
        for j in range(self.y):
            for i in range(self.x):
                if current[i][j] == 0:
                    unmarked.append((i, j))
                elif current[i][j] == -3:
                    marked_mine.append((i, j))
        return unmarked, marked_mine

    def print_current_board(self):
        # Print the current board status
        # _       : Unassigned
        # |       : Assigned no mine
        # *       : Assigned mine
        # [0-8] : Hint
        current = copy.deepcopy(self.marked)
        for j in range(self.y):
            for i in range(self.x):
                current[i][j] = '_' if current[i][j] == 0 else current[i][j]
                current[i][j] = self.hints[i][j] if current[i][j] == -2 else current[i][j]
                current[i][j] = '*' if current[i][j] == -3 else current[i][j]
                print(current[i][j], end=" ")
            print()

    def print_answer_board(self):
        # Print the answer board
        # *       : Mine
        # [0-8] : Hint
        board = copy.deepcopy(self.hints)
        for j in range(self.y):
            for i in range(self.x):
                board[i][j] = '*' if board[i][j] == -3 else board[i][j]
                print(board[i][j], end=" ")
            print()


if __name__ == '__main__':
    # Examples
    b = Board('easy')
    b.print_answer_board()
```

test.py

```
from minesweeper import MineSweeper, Result
```

```
def simple_test(rounds, difficulty, show = False):
    success = 0
    fail = 0
    stuck = 0
    success_duration = 0

    for i in range(rounds):
        game = MineSweeper(difficulty)
        result = game.play(show)
        if result.status == 'Success':
            success += 1
            success_duration += result.play_time
        elif result.status == 'Fail':
            fail += 1
        if result.stuck:
            stuck += 1

    print()
    print('==============')
    print('Tested:\t\t\t', rounds, difficulty, 'games')
    print('Success:\t\t', success, 'games')
    print('Success duration:\t', success_duration/success, 'sec
per game')
    print('Fail (Stuck):\t\t {} ({}) games'.format(fail, stuck))

def mines_count_test(rounds, difficulty, mines, show = False):
    print('==============')
    print('Tested:', rounds, difficulty, 'games per mines_counts')
    print('Mines\tSuccess\tStuck\tFail\tFail-Stuck\tSuccess
duration')
    for m in range(1, mines+1):
        success = 0
        fail = 0
        stuck = 0
        success_duration = 0

        for i in range(rounds):
            game = MineSweeper(difficulty, m)
            result = game.play(show)
            if result.status == 'Success':
                success += 1
                success_duration += result.play_time
            elif result.status == 'Fail':
                fail += 1
            if result.stuck:
                stuck += 1

        print('{}\t{}\t{}\t{}\t{}\t\t{}'.format(m, success, stuck,
fail, fail-stuck, success_duration/success))
```

```
if __name__ == '__main__':
    simple_test(1000, 'easy')
    simple_test(1000, 'medium')
    simple_test(1000, 'hard')
    mines_count_test(100, 'medium', 40)
```