Homework 3 Report

0516034 楊翔鈞

#1 作業介紹

作業題目為利用邏輯推導的方式讓電腦解踩地雷遊戲，盤面中的每個 cell 只會有兩種可能的值，True 代表是地雷、False 代表是安全，遊戲的一開始會先在 knowledge base 裡面放幾個 initial safe cell，隨著這些 safe cell 被標記，跟該 cell 有關的 clause 就會被加到 knowledge base 裡面，當無法從 knowledge base 裡面的找出任何 single-literal clause 時（無法直接標記任何 cell），則對整個 knowledge base 做 resolution，看是否能夠產生新的 clause，而遊戲的中止條件為「所有 cell 都被標記」或是「遊戲沒辦法繼續進行下去（做完 resolution 仍沒辦法標出任何 cell）」。

對於 easy、medium、hard 三種難度的盤面大小以及地雷數量如下：

- easy
  - 9 x 9 = 81 個 cell
  - 10 顆地雷
- medium
  - 16 x 16 = 256 個 cell
  - 25 顆地雷
- hard
  - 16 x 30 = 480 個 cell
  - 99 顆地雷

#2 實作方法

我用 string 表示 literal，例如 $cell_{1,2}$ 代表第一列、第二行的 cell，用 string 表示就會變成 $"1,2"$，negative 的話就是 $"not\ 1,2"$；clause 則是由一個至多個 string 組成的 list，例如 $cell_{1,2} \vee cell_{1,3} \vee cell_{1,4}$ 就會是 $["1,2", "1,3", "1,4"]$。另外 global constraint 是當未被標記的 cell 數量小於 10 的時候才會被加到 knowledge base 中。

#3 實驗

對於三個不同的難度，我嘗試了三種不同的 initial safe cell 數量，分別是 $\frac{\sqrt{\#cells}}{2}$ 、 $\sqrt{\#cells}$ 、 $2\sqrt{\#cells}$，結果如下表，其中 matching 次數代表無法在 knowledge base 中找 single-literal clause，而對整個 knowledge base 做 pairwise matching 的次數。

#3.1 easy

initial safe cell 數量	marked cell 數量	matching 次數
4	4	2
4	81	0
4	81	1
9	79	1
9	81	0
9	81	0
18	81	0
18	81	0

18	81	0

#3.2 medium

initial safe cell 數量	marked cell 數量	matching 次數
8	256	0
8	256	0
8	256	0
16	256	0
16	256	0
16	256	0
32	254	1
32	256	0
32	256	0

#3.3 hard

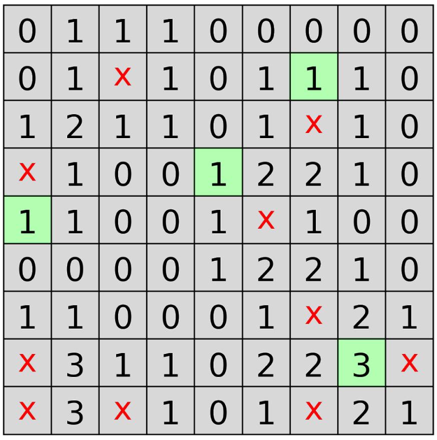
initial safe cell 數量	marked cell 數量	matching 次數
11	355	5
11	450	4
11	460	4
22	434	5
22	476	4
22	480	5
44	472	3
44	480	2
44	480	3

#4 觀察

#4.1 marked cell 數量

選到 hint 不等於 0 的 cell 基本上只會增加 knowledge base 裡面 multiple-literal clause 的數量，而遊戲的進行通常都仰賴於 single-literal clause，因為 single-literal clause 可以讓我們直接標記 cell，進而去修改 knowledge base 中的 multiple-literal clause。

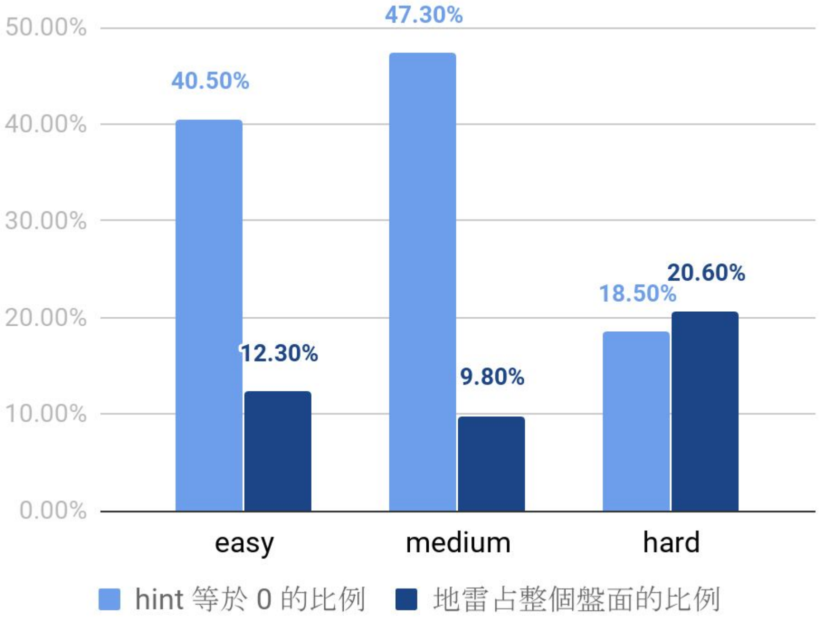
因為一開始選擇 initial safe cell 的時候是隨機挑選的，如果 initial safe cell 的數量不多，而且又剛好都選到 hint 不等於 0 的 cell，就很有可能會解不下去，就像上面 easy 的表格中，有一筆是 4 個 initial safe cell，然後只標了 4 個 cell 就卡住，以下為該實驗的結束時的盤面，灰色代表未標記；綠色是標記為安全；黃色是標記為地雷：



0	1	1	1	0	0	0	0	0
0	1	x	1	0	1	1	1	0
1	2	1	1	0	1	x	1	0
x	1	0	0	1	2	2	1	0
1	1	0	0	1	x	1	0	0
0	0	0	0	1	2	2	1	0
1	1	0	0	0	1	x	2	1
x	3	1	1	0	2	2	3	x
x	3	x	1	0	1	x	2	1

#4.2 matching 次數

在比較 matching 次數前，我想先比較三種難度 hint 等於 0 的比例以及地雷與安全 cell 的比例，對於這三種難度，我各隨機產生了 3000 個盤面，並統計其數據，結果如下：



可以發現 hard 的地雷的比例最高；medium 的最低，這也反映到 hint 等於 0 的比例上，medium 的比例最高；hard 的最低，而我認為這跟 matching 次數有直接的關係，如果 hint 等於 0 的比例高，代表在標記的過程中能夠產生較多的 single-literal clause，而非一大堆沒辦法直接解開的 multiple-literal clause，在上面比較 marked cell 數量的時候也說過，越多的 single-literal clause 對於解開整個盤面是越有利的，在把 single-literal clause 標記起來的同時，可以利用這些已經確認的 cell 去更新 knowledge base 裡面的 multiple-literal clause（縮短或是從 knowledge base 中移除），也就比較不會出現 knowledge base 裡面沒有 single-literal clause，然後得透過 matching 產生新 clause 的情況。

從上面的長條圖可以看到 easy 跟 medium 的 hint 等於 0 的比例很高，對應到上個部分的實驗數據表格，easy 跟 medium 的 matching 次數確實比 hard 還要少，而且仔細觀察可以發現 meduim 的次數又比 easy 的略少一點，這也對應到長條圖中 easy 跟 medium 兩者的差距，medium 的 hint 等於 0 的比例比 easy 的多了幾個百分點。

#5 Optional / Extra Credits

- How to use first-logic here?
  把每個 cell 都當成一個物件，Mine($cell_{i,j}$)代表 $cell_{i,j}$ 是地雷，Safe($cell_{i,j}$)代表 $cell_{i,j}$ 是安全的，另外再建立兩種 relation：AtLeastOneMine 跟 AtLeastOneSafe，用來表示選到

hint 不等於 0 時產生的那些 clause，例如 $cell_{1,2} \lor cell_{1,3} \lor cell_{1,4}$ 就可以表示成 AtLeastOneMine($cell_{1,2}$, $cell_{1,3}$, $cell_{1,4}$)。

- Discuss whether forward chaining or backward chaining applicable to this problem.
    - forward checking：可以。用 initial safe cell 去推導鄰近的格子的狀態，然後再用推導出來的結果繼續延伸。
    - backward checking：沒辦法，一開始知道的資訊太少。如果我們推測某個 cell 是地雷，就必須回推怎樣的情況會讓這個 cell 是地雷，但是一開始手上的資訊只有 initial safe cell，沒有足夠的資訊可以讓我們去推導。

- Propose some ideas about how to improve the success rate of "guessing" when you want to proceed from a "stuck" game.
  亂猜，我認為除了亂猜以外沒有任何可行的方法，用暴力搜尋也只會找到多個可能的解，沒辦法確定哪一個才是符合該盤面的，就算改用類神經網路去學習可能也無法得到很好的結果，因為地雷都是隨機分佈，並沒有一個很具體的行為模式可以學習。

- Discuss ideas of modifying the method in Assignment#2 to solve the current problem.
  把 initial safe cell 裡面的 cell 都放進 unassign list 裡面，然後把他們的 domain 都設成 0 ，contraint set 一開始則是空的，然後：
    1. 用 MRV 跟 degree heuristic 挑選下一個要 assign 的 cell
    2. 如果選到的 cell 不是地雷，就把跟該 cell 有關的 contraint 加到 constraint set 裡面
    3. 做 forward checking 更新每個 unassigned cell 的 domain

  重複上面這些步驟直到所有 cell 都被標記而且沒有標錯的情況。

```
import os
import time
import numpy as np
import matplotlib.pyplot as plt
from itertools import combinations as comb
from collections import OrderedDict

MINE = 100

OPEN = 200
FLAG = 300

board_meta = {
    'easy': {
        'board_size': (9, 9),
        'mine_num': 10,
        'annot_size': 20
    },
    'medium': {
        'board_size': (16, 16),
        'mine_num': 25,
        'annot_size': 10
    },
    'hard': {
        'board_size': (16, 30),
        'mine_num': 99,
        'annot_size': 8
    }
}


# plot the board with the answer and current status
# just for visualization
def plot_board(status, ans, annot_size):
    l, w = status.shape
    args_mine = OrderedDict(
        color='r',
        fontsize=annot_size,
        horizontalalignment='center',
        verticalalignment='center')
    args_hint = OrderedDict(
        fontsize=annot_size,
        horizontalalignment='center',
        verticalalignment='center')

    for i in range(l):
        for j in range(w):
            if status[i, j] == -1:
                plt.fill([j, j, j+1, j+1], [l-i-1, l-i, l-i, l-i-1],
c='gray', alpha=0.3)
```

```
            elif status[i, j] == FLAG:
                plt.fill([j, j, j+1, j+1], [l-i-1, l-i, l-i, l-i-1],
c='yellow', alpha=0.3)
            elif status[i, j] == OPEN:
                plt.fill([j, j, j+1, j+1], [l-i-1, l-i, l-i, l-i-1],
c='lime', alpha=0.3)
            if ans[i, j] == MINE:
                plt.annotate('x', (j+0.5, l-i-0.5), **args_mine)
            else:
                plt.annotate(ans[i, j], (j+0.5, l-i-0.6), **args_hint)

    plt.gca().set_xticks(np.arange(w+1))
    plt.gca().set_yticks(np.arange(l+1))
    plt.gca().set_xlim([0, w])
    plt.gca().set_ylim([0, l])
    plt.gca().tick_params(bottom=False, top=False, left=False,
right=False)
    plt.gca().tick_params(labelbottom=False, labeltop=False,
labelleft=False, labelright=False)
    plt.gca().grid(b=True, which='major', color='k', linestyle='-')
    plt.gca().set_aspect('equal')


# generate a board with given size and number of mines randomly
# return the answer and initial safe cells
def generate_board(board_size, mine_num, init_safe_cell_num):
    mines = set()
    while len(mines) < mine_num:
        i = np.random.randint(0, board_size[0])
        j = np.random.randint(0, board_size[1])
        mines.add((i, j))
    ans = np.zeros((board_size[0]+2, board_size[1]+2), dtype=np.int32)
    for mine in mines:
        ans[mine[0]+1, mine[1]+1] = MINE
        i, j = mine[0]+1, mine[1]+1
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                new_i = i + di
                new_j = j + dj
                if ans[new_i, new_j] == MINE:
                    continue
                ans[new_i, new_j] += 1
    ans = ans[1:-1, 1:-1]
    safe_cell = np.argwhere(ans!=MINE)
    idx = np.random.choice(np.arange(0, safe_cell.shape[0]),
init_safe_cell_num, replace=False)

    return ans, safe_cell[idx]
```

```
# get the eight neighbors of given cell
def get_neighbors(board_size, i, j):
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di==0 and dj==0:
                continue
            new_i = i + di
            new_j = j + dj
            if (new_i<0 or new_i>=board_size[0] or
                    new_j<0 or new_j>=board_size[1]):
                continue
            neighbors.append(f'{new_i},{new_j}')
    return neighbors


# get all unmarked cells
def get_unmark_cells(board_size, KB0):
    cells = []
    for i in range(board_size[0]):
        for j in range(board_size[1]):
            if f'{i},{j}' not in KB0:
                cells.append(f'{i},{j}')
    return cells


# check whether the two clauses are identical
def duplicate_pairwise(clause1, clause2):
    match = 0
    for literal in clause1:
        if literal in clause2:
            match += 1
    if match==len(clause2) and len(clause1)==len(clause2):
        return True
    return False


# check whether there exists any clause identical to the given clause in
the
# knowledge base
def duplicate(KB, clause):
    for sentence in KB:
        if duplicate_pairwise(clause, sentence):
            return True
    return False


# subsumption for single-literal clause
def subsumption_single(KB, clause):
    for i in range(len(KB)):
        if clause in KB[i]:
            KB[i] = []


# subsumption between two clauses
def subsumption_pairwise(KB, i, j):
    clause1, clause2 = KB[i], KB[j]
    match = 0
    for literal in clause1:
        if literal in clause2:
            match += 1
    if match==len(clause1) and len(clause1)<len(clause2):
        KB[j] = []
        return True
    elif match==len(clause2) and len(clause2)<len(clause1):
        KB[i] = []
        return True
    return False


# subsumption for given clause to the knowledge base
def subsumption(KB, clause):
    insert = True
    update = False
    for i in range(len(KB)):
        sentence = KB[i]
        if not len(sentence):
            continue
        match = 0
        for literal in clause:
            if literal in sentence:
                match += 1
        if match==len(sentence) and len(sentence)<=len(clause):
            insert = False
        elif match==len(clause) and len(sentence)>len(clause):
            KB[i] = []
            update = True
    if insert:
        KB.append(clause)
        update = True
    return update


# generate new clause if there is only one pair on complementary
literals
# between two clauses
def comp(clause1, clause2):
    c1 = list(clause1)
    c2 = list(clause2)
    match = 0
    for literal in clause1:
        if 'not' in literal:
            l = literal[4:]
        else:
            l = f'not {literal}'
        if l in clause2:
            c1.remove(literal)
            c2.remove(l)
            match += 1
    if match == 1:
        return list(set(c1+c2))
    else:
        return []


if __name__ == '__main__':
    level = 'hard'
    meta = board_meta[level]
    board_size = meta['board_size']
    mine_num = meta['mine_num']
    annot_size = meta['annot_size']
    cell_num = board_size[0] * board_size[1]
    unmark_mine_num = mine_num
    unmark_cell_num = cell_num
    init_safe_cell_num = int(np.round(np.sqrt(cell_num)))
    ans, init_safe_cell = generate_board(board_size, mine_num,
init_safe_cell_num)

    folder = f'{level}_{init_safe_cell_num}_{int(time.time())}'
    os.mkdir(folder)

    KB0 = {}
    KB = []
    # add initial safe cells into knowledge base
    for cell in init_safe_cell:
        clause = ','.join(cell.astype(str))
        clause = f'not {clause}'
        KB.append([clause])

    print(init_safe_cell)
    # initialize the status
    # -1 for unmarked
    # OPEN for marked as safe
    # FLAG for marked as mine
    status = np.ones(ans.shape, dtype=np.int32) * -1
    cnt, not_found = 0, 0
    # while knowledge base is not empty
    while KB:
        # if the number of unmarked cells is less than 10, add the
global
```

```
        # constaints to knowledge base
        if unmark_cell_num <= 10:
            unmark_cells = get_unmark_cells(board_size, KB0)
            for clause in list(comb(unmark_cells,
unmark_cell_num-unmark_mine_num+1)):
                clause = list(clause)
                if not duplicate(KB, clause):
                    subsumption(KB, clause)
            for clause in list(comb(unmark_cells, unmark_mine_num+1)):
                clause = list(clause)
                for i in range(len(clause)):
                    clause[i] = f'not {clause[i]}'
                if not duplicate(KB, clause):
                    subsumption(KB, clause)
        KB = sorted(KB, key=lambda t: len(t))
        found = False
        for i in range(len(KB)):
            clause = KB[i]
            # look for a single-literal clause
            if len(clause) == 1:
                found = True
                print(cnt, clause[0])
                # this cell is safe, update the status (OPEN)
                if clause[0].startswith('not'):
                    # put the marked cell into KB0
                    KB0[clause[0][4:]] = False
                    # remove that literal from knowledge base
                    KB[i] = []
                    unmark_cell_num -= 1
                    subsumption_single(KB, clause[0])
                    for j in range(len(KB)):
                        if clause[0][4:] in KB[j]:
                            KB[j].remove(clause[0][4:])
                    cell = clause[0][4:].split(',')
                    r, c = int(cell[0]), int(cell[1])
                    status[r, c] = OPEN
                    hint = ans[r, c]
                    neighbors = get_neighbors(board_size, r, c)
                    for j in range(len(neighbors)):
                        # only consider the unmarked cells
                        if neighbors[j] in KB0:
                            if KB0[neighbors[j]] == True:
                                hint -= 1
                            neighbors[j] = ''
                    neighbors = [n for n in neighbors if len(n) > 0]
                    # m = number of unmarked neighbors
                    # n = hint
                    # (m == n): insert the m single-literal positive
clauses
                    # to the knowledge base, one for each unmarked
```

```
neighbor
                    if hint == len(neighbors):
                        for n in neighbors:
                            if not duplicate(KB, n):
                                KB.append([n])
                    # (n == 0): insert the m single-literal negative
clauses
                    # to the knowledge base, one for each unmarked
neighbor
                    elif hint == 0:
                        for n in neighbors:
                            if not duplicate(KB, f'not {n}'):
                                KB.append([f'not {n}'])
                    # (m > n > 0): generate CNF clauses and add them to
the
                    # knowledge base
                    elif len(neighbors) > hint:
                        for clause in list(comb(neighbors,
len(neighbors)-hint+1)):
                            clause = list(clause)
                            if not duplicate(KB, clause):
                                subsumption(KB, clause)
                        for clause in list(comb(neighbors, hint+1)):
                            clause = list(clause)
                            for i in range(len(clause)):
                                clause[i] = f'not {clause[i]}'
                            if not duplicate(KB, clause):
                                subsumption(KB, clause)
                    else:
                        print(f'ERROR: hint: {hint}, neighbors:
{len(neighbors)}')
                # this cell is mine, update the status (FLAG)
                else:
                    # put the marked cell into KB0
                    KB0[clause[0]] = True
                    # remove that literal from knowledge base
                    KB[i] = []
                    unmark_cell_num -= 1
                    unmark_mine_num -= 1
                    subsumption_single(KB, clause[0])
                    for j in range(len(KB)):
                        if f'not {clause[0]}' in KB[j]:
                            KB[j].remove(f'not {clause[0]}')
                    cell = clause[0].split(',')
                    r, c = int(cell[0]), int(cell[1])
                    status[r, c] = FLAG
                # plot the current status
                plot_board(status, ans, annot_size)
                plt.savefig(f'{folder}/{cnt}.png', dpi=300,
transparent=True)
```

```
                plt.clf()
                cnt += 1
                break
        # if we cannot found a single-literal clause
        if not found:
            print('Pairwise matching')
            update = False
            not_found += 1
            for i in range(len(KB)):
                for j in range(i+1, len(KB)):
                    if len(KB[i])==0 or len(KB[j])==0:
                        continue
                    # check whether the two clauses are identical
                    if not duplicate_pairwise(KB[i], KB[j]):
                        if not subsumption_pairwise(KB, i, j):
                            if len(KB[i])>2 and len(KB[j])>2:
                                continue
                            new_clause = comp(KB[i], KB[j])
                            if new_clause:
                                if not duplicate(KB, new_clause):
                                    if subsumption(KB, new_clause):
                                        update = True
                        else:
                            update = True
                    else:
                        KB[j] = []
                        update = True
            # stop if the knowledge base didn't update after pairwise
matching
            # (can neither find a single-literal clause nor generate any
new
            # clause from the knowkedge base)
            if not update:
                break
        KB = [t for t in KB if len(t) > 0]
    print('--------------------')
    print(f'{cnt} cells marked')
    print(f'Pairwise matching: {not_found} times')
    os.rename(folder, f'{folder} {cnt} {not_found}')
```