Introduction to Artificial Intelligence Prog. 3 Report

0516003 李智嘉

1. Introduction

- 在這份報告，我們將利用 Logical inference 來玩踩地雷遊戲
- 遊戲一開始，會直接給定玩家初始一定數量的安全格子，玩家將利用此列表來解出遊戲
- 我們定義：“遊戲成功” 為順利完所有線索，推得最大可能的盤面（也就是說剩下的格子可能有複數種下法，我們只能去猜測，不能由邏輯推得正解），“遊戲失敗” 為無法利用完所有的線索來取得答案。
- 以下我將針對下列事項來進行實驗
  - 簡單盤面（9 by 9 board），不同地雷數，對成功率的影響
  - 中等盤免（16 by 16 board），不同地雷數，對成功率的影響
  - 困難盤面（30 by 16 board），不同地雷數，對成功率的影響
  - 三種模式（9 by 9 with 10 mines, 16 by 16 with 25 mines, 30 by 16 with 99 mines），需達到90%以上成功率，最少需多少初始安全格子

2. Experiment

- 簡單盤面（9 by 9 board），不同地雷數，對成功率的影響
  - 我Sample 100筆不同的棋盤，初始皆是給 9 格安全格子，根據不同地雷個數，統計其成功與失敗個數

地雷數	成功個數	失敗個數	花費時間（秒）
4	100	0	0.0115
6	98	2	0.0247
8	90	10	0.0391
10	79	21	0.0557
12	59	41	0.1937
14	37	63	0.6654
16	20	80	0.7406
18	3	97	1.0137

  - 從上表可以看到，隨著地雷數增長，成功率明顯的降低
  - 原因可能是因為，地雷數越多，我們得到的資訊就越複雜，要知道某格的狀況所要推論的東西也就越多，不能很直接的知道某個格子有沒有地雷，而這也相當符合人實際在玩遊戲的情況
  - 而在實際遊戲難度（10個地雷），我們可以拿到 79% 的成功率

- 中等盤免（16 by 16 board），不同地雷數，對成功率的影響
  - Sample 100筆不同的棋盤，初始皆是給 16 格安全格子，根據不同地雷個數，統計其成功與失敗個數

地雷數	成功個數	失敗個數	花費時間（秒）
19	96	4	0.1527
21	90	10	0.1500
23	87	13	0.1769
25	86	14	0.2128
27	83	17	0.2279
29	79	21	0.3698
31	82	18	0.3558

  - 隨著地雷數成長，成功率一樣逐漸降低，但這次下降的沒有簡單盤面的快，也許是初始格子較多的原因
  - 這次在實際遊戲的中等難度（25個地雷）的成績也比簡單難度還高，有 86% 的成功率
  - 很特別的，在地雷數29時，失敗率比地雷數31還要高。但我覺得只是sample的誤差
  - 可以很有趣的看到，即使是16*16的盤面，在31顆地雷的情況下，平均解一個盤面的速度與成功率，都比簡單盤面在14顆以上的地雷還要快且準。這可能也代表了初始安全地雷的個數有多重要

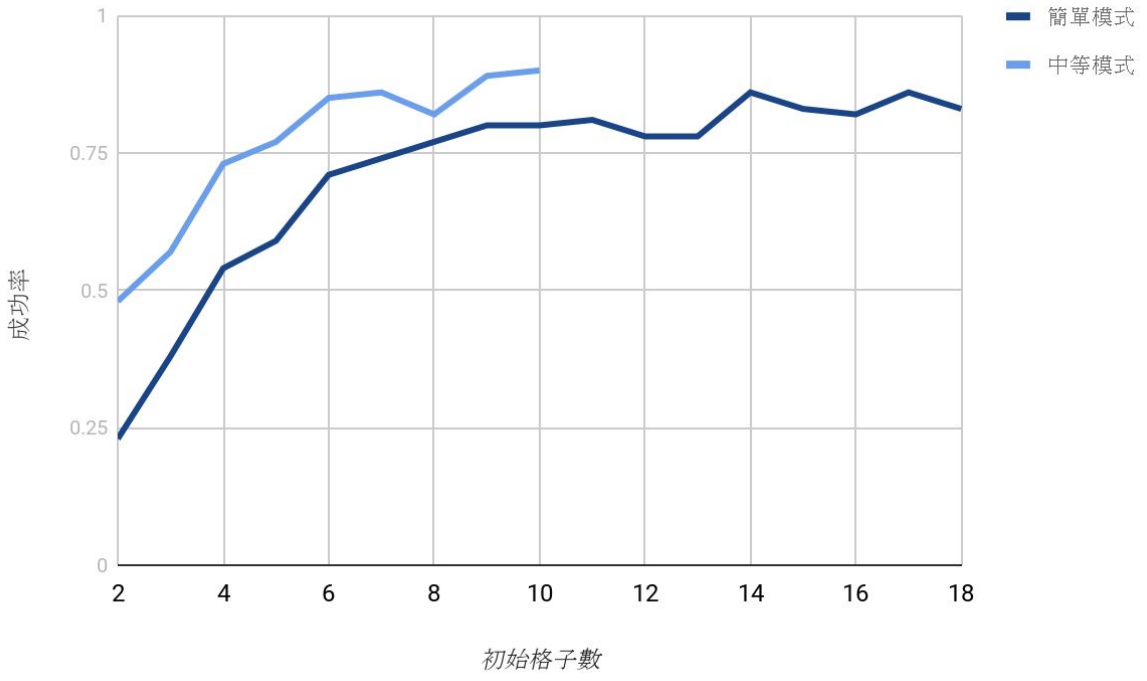
- 困難盤面（30 by 16 board），不同地雷數，對成功率的影響
  - Sample 100筆不同的棋盤，初始皆是給 22 格安全格子，根據不同地雷個數，統計其成功與失敗個數

地雷數	成功個數	失敗個數	花費時間（秒）
50	89	11	0.7229
55	72	28	0.7905
60	64	36	1.040
65	62	38	0.9710
70	52	48	1.2579
75	28	72	1.2560
80	26	74	2.4456

85	15	85	3.2925

- 困難盤面也是隨著地雷數增加，要花的時間也增加，成功率也跟著下降
- 這次甚至還沒到實際困難難度（99個地雷），在85個地雷時，成功率就只剩下15%了
- 綜合上面的數據，我覺得可能以 sqrt(cells number) 當作初始安全格子，在這邊並不能很好的推論盤面
- 另一個原因也可能是因為，目前我生成初始安全格子的方法，是非常全盤uniform的去選，這可能會讓我們初始擁有的資訊彼此不那麼靠近，因此陷入沒有東西可以再推論出來而失敗。所以當盤面越大，對於 AI 來說也就越難推出解答

- 三種模式達到90%以上成功率，最少需多少初始安全格子



- 上圖是針對兩種模式，根據不同初始格子數去sample 100局遊戲的成功率
- 另我驚訝的是，中等模式在同樣初始格子數的情況下，成功率竟然比簡單模式還要高。某種程度上代表了中等模式比簡單模式還要容易
- 原因可能是因為地雷數與棋盤大小的比例。簡單模式的地雷與總格子比是0.123 (10/81)，中等模式是0.098 (25/256)，可以看到中等模式的比值反而是比簡單模式低的，也就是他只需要比較少的資訊就可以完成遊戲

    - 而困難模式的地雷與總格子比是0.206 (99/480)，經過同樣的實驗，困難模式需要初始格子到285才可以到達0.9以上
    - 但會這麼多的一個原因也是因為，如同我們上面提到過得，這邊初始格子的選擇是全盤uniform的去挑，可能格子不會互相靠近，AI也就無法從初始格子推得更多訊息
    - 所以這邊的實驗結果應該是可以代表，人在玩踩地雷時，“最多”應該只需要猜測這些數量的格子，就可以推論出整盤遊戲

3. Conclusion and remaining question

- 這項作業我們利用了 Logical inference 的方法來玩踩地雷遊戲
- 根據實驗結果，我們發現地雷數越多，遊戲的難度也越高，也在官方設定的三種模式中，中等模式反而比簡單模式還要容易完成，只是需要的時間比較久
- 這項實驗中，我在選取初始安全格子時，是全盤uniform的去挑選。可能會造成初始格子彼此不接近，導致不能推論出更多資訊，讓遊戲失敗。這點與人類實際玩遊戲有很大的不同。如果有時間的話，也許可以改成是選取一大塊連續的初始格子，或是遊玩過程中動態的給定安全格子，最後再計算需要給幾次才會成功

4. Further discussion

- Discuss ideas of modifying the method in Assignment#2 to solve the current problem.
    - 利用作業二的方法在踩地雷的話，這就會變成constraint也會隨著expand的node而改變的問題
    - 一開始一樣會有初始的格子，透過這些初始格子我們就可以有初始的constraint
    - 之後就是配合不同heuristic與forward checking，來挑選未知的格子以及遍歷這格子的domain
    - 在expand一個node的時候，會出現兩種狀況。一種是assign錯誤的值，他其實是地雷但AI assign成安全的，這邊就會直接失敗。一種是安全的，可能AI assign成地雷，或是assign成安全的且他也真的安全。在後者的情況我們可能會增加constraint，這也要跟著加進目前的constraint裡面
    - 以上就是利用Constraint Satisfaction的方法來做這件事的說明，可以看到他其實有可能會直接失敗，所以可能heuristic就要好好挑選。我猜測MRV應該會表現最好，因為剩下的domain就是符合當前constraint的值，先assign肯定沒錯

```
import random
import itertools

class Environment():
    def __init__(self, x_size, y_size, mines_num):
        self.board = []
        self.x_size = x_size
        self.y_size = y_size
        self.mines_num = mines_num

        ## initial board
        for _ in range(y_size):
            self.board.append([])
            for _ in range(x_size):
                self.board[-1].append([False, 0])   ## [is_mine,
mines_num_around_it]

        ## create mines
        done_mine_num = 0
        while done_mine_num != mines_num:
            x, y = random.randint(0, x_size-1), random.randint(0,
y_size-1)
            if not self.board[y][x][0]:
                self.board[y][x] = [True, -1]
                done_mine_num += 1

        ## update safe cell number
        for y in range(y_size):
            for x in range(x_size):
                if not self.board[y][x][0]:
                    self.board[y][x][1] = self.get_sur_mines_num(x, y)

    ## get surrounding mines number
    def get_sur_mines_num(self, x_idx, y_idx):
        mines_cnt = 0
        for dy in [-1, 0, 1]:
            if y_idx+dy < 0 or y_idx+dy >= self.y_size:
                continue
            new_y = y_idx+dy
            for dx in [-1, 0, 1]:
```

```
                if x_idx+dx < 0 or x_idx+dx >= self.x_size:
                    continue
                new_x = x_idx+dx
                if self.board[new_y][new_x][0]:
                    mines_cnt += 1
        return mines_cnt

    ## display whole board
    def show_whole_board(self):
        for y in range(self.y_size):
            for x in range(self.x_size):
                if self.board[y][x][0]:
                    print('X', end=', ')
                else:
                    print(self.board[y][x][1], end=', ')
            print()

    ## give safe cells list (for initial player)
    def give_safe_cells(self, safe_num):
        done_num = 0
        safe_list = []  ## element: (x_idx, y_idx, sur_mines_num)
        # x, y = random.randint(0, self.x_size-1), random.randint(0,
self.y_size-1)
        while done_num != safe_num:
            x, y = random.randint(0, self.x_size-1), random.randint(0,
self.y_size-1)
            if (not self.board[y][x][0]) and ((x, y,
self.board[y][x][1]) not in safe_list):
                safe_list.append((x, y, self.board[y][x][1]))
                done_num += 1
            # dx, dy = random.randint(-1, 1), random.randint(-1, 1)
            # if x+dx >= 0 and x+dx < self.x_size:
            #     x = x+dx
            # if y+dy >= 0 and y+dy < self.y_size:
            #     y = y+dy

        return safe_list

class Agent():
    def __init__(self, x_size, y_size, mines_num, safe_cell_list):
```

```
        self.board = []
        self.x_size = x_size
        self.y_size = y_size
        self.mines_num = mines_num
        self.KB0 = dict()
        self.KB = []    ## element: [[Pos_or_Neg(True/False), x_idx,
y_idx], ...]

        self.neighbor = []

        ## initial player's board and neighbor list for every cell
        for y_idx in range(y_size):
            self.board.append([])
            self.neighbor.append([])
            for x_idx in range(x_size):
                self.board[-1].append([False, -1])   ## [has_assigned,
assigned_value(False means not mine)]
                neig = []
                for dy in [-1, 0, 1]:
                    if y_idx+dy < 0 or y_idx+dy >= self.y_size:
                        continue
                    for dx in [-1, 0, 1]:
                        if x_idx+dx < 0 or x_idx+dx >= self.x_size or
(dx==0 and dy==0):
                            continue
                        neig.append((x_idx+dx, y_idx+dy))
                self.neighbor[-1].append(neig)

        ## put initial safe cell
        self.init_safe_cell(safe_cell_list)

    ## display the board the user currently know
    def show_current_board(self):
        print('============================')
        for y_idx in range(self.y_size):
            for x_idx in range(self.x_size):
                if (x_idx, y_idx) in self.KB0.keys():
                    if self.board[y_idx][x_idx][1] < 0:
                        print('X', end=', ')
                    else:
```

```
                    print(self.board[y_idx][x_idx][1], end=', ')
                else:
                    print('-', end=', ')
            print()

    ## check sen2 contain sen1, for subsumption
    def all_in(self, sen1, sen2):
        success = True
        for element1 in sen1:
            for element2 in sen2:
                if not (element1 == element2):
                    success = False
                    break
            if not success:
                break
        return success

    ## insert a sentence to KB
    def insert_KB(self, sentence):
        if len(sentence) > 1:
            del_list = []
            sentence_len = len(sentence)

            ## Apply KB0 knowledge to shrink sentence
            for idx in range(sentence_len):
                if (sentence[idx][1], sentence[idx][2]) in
self.KB0.keys():
                    if (self.KB0[(sentence[idx][1], sentence[idx][2])]
and sentence[idx][0])\
                        or (not self.KB0[(sentence[idx][1],
sentence[idx][2])] and not sentence[idx][0]):
                        return
                    else:
                        del_list.append(idx)
            del_list.reverse()
            self.check_dec_order(del_list)
            for idx in del_list:
                del sentence[idx]

            ## Check subsumption
```

```
            del_list = []
            for idx in range(len(self.KB)):
                if sentence_len < len(self.KB[idx]) and
self.all_in(sentence, self.KB[idx]):
                    del_list.append(idx)
                elif sentence_len > len(self.KB[idx]) and
self.all_in(self.KB[idx], sentence):
                    return
            del_list.reverse()
            for idx in del_list:
                del self.KB[idx]

        ## if the sentence has only one clause, and it exist in KB0,
don't insert it
        elif (sentence[0][1], sentence[0][2]) in self.KB0.keys():
            return

        if sentence not in self.KB:
            self.KB.append(sentence)

    ## get clause for difference cases
    def gen_clause(self, x_idx, y_idx, mines_num):
        neig_num = len(self.neighbor[y_idx][x_idx])
        if mines_num == 0:
            for neig in self.neighbor[y_idx][x_idx]:
                self.insert_KB([[False, neig[0], neig[1]]])
        elif neig_num == mines_num:
            for neig in self.neighbor[y_idx][x_idx]:
                self.insert_KB([[True, neig[0], neig[1]]])
        else:
            true_elements = []
            for neig in self.neighbor[y_idx][x_idx]:
                true_elements.append([True, neig[0], neig[1]])

            false_elements = []
            for neig in self.neighbor[y_idx][x_idx]:
                false_elements.append([False, neig[0], neig[1]])

            pos_literals = list(itertools.combinations(true_elements,
neig_num-mines_num+1))
```

```
            neg_literals = list(itertools.combinations(false_elements,
mines_num+1))

            for sen in pos_literals:
                self.insert_KB(list(sen))
            for sen in neg_literals:
                self.insert_KB(list(sen))

    ## initial player's safe cell list
    def init_safe_cell(self, safe_list):
        for safe_cell in safe_list:
            self.KB0[(safe_cell[0], safe_cell[1])] = False
            self.board[safe_cell[1]][safe_cell[0]][0] = True
            self.board[safe_cell[1]][safe_cell[0]][1] = safe_cell[2]

            self.gen_clause(*safe_cell)

    ## get a the sentence index which could do resolution with other
sentence
    def get_resolv_pairs(self):
        len_KB = len(self.KB)
        for sen_idx in range(len_KB):
            for sen_idx2 in range(sen_idx+1, len_KB):
                dup, fail = False, False
                for idx in range(len(self.KB[sen_idx])):
                    for idx2 in range(len(self.KB[sen_idx2])):
                        if self.KB[sen_idx][idx][1] ==
self.KB[sen_idx2][idx2][1] and\
                            self.KB[sen_idx][idx][2] ==
self.KB[sen_idx2][idx2][2] and\
                            self.KB[sen_idx][idx][0] !=
self.KB[sen_idx2][idx2][0]:
                            if dup:
                                fail = True
                                break
                            else:
                                dup = True
                                break
                    if fail:
                        break
```

```
                if not fail and dup:
                    return sen_idx
        return None

    ## do resolution
    def matching(self):
        sen_idx = self.get_resolv_pairs()

        del_list = []
        new_sentence_list = []
        if sen_idx is not None: ## if we have cell which could do
resolution
            base_sentence = self.KB[sen_idx]
            del self.KB[sen_idx]

            ## iterate every sentence in our KB
            for sen_idx2 in range(len(self.KB)):
                dup_sen_idx1, dup_sen_idx2 = -1, -1
                fail = False
                for idx in range(len(base_sentence)):
                    for idx2 in range(len(self.KB[sen_idx2])):
                        if base_sentence[idx][1] ==
self.KB[sen_idx2][idx2][1] and\
                            base_sentence[idx][2] ==
self.KB[sen_idx2][idx2][2] and\
                            base_sentence[idx][0] !=
self.KB[sen_idx2][idx2][0]:
                            if dup_sen_idx1 != -1:
                                fail = True
                                break
                            else:
                                dup_sen_idx1, dup_sen_idx2 = idx, idx2
                                break
                    if fail:
                        break

                ## doing resolution
                if not fail and dup_sen_idx1 != -1:
                    new_sentence = []
                    for idx in range(len(base_sentence)):
```

```
                        if idx != dup_sen_idx1:
                            new_sentence.append(base_sentence[idx])
                    for idx in range(len(self.KB[sen_idx2])):
                        if idx != dup_sen_idx2 and
self.KB[sen_idx2][idx] not in new_sentence:
                            new_sentence.append(self.KB[sen_idx2][idx])

                    new_sentence_list.append(new_sentence)
                    del_list.append(sen_idx2)

        ## delete the old sentences
        del_list.reverse()
        self.check_dec_order(del_list)
        for idx in del_list:
            del self.KB[idx]

        ## insert the new sentences
        for sentence in new_sentence_list:
            self.insert_KB(sentence)

    ## check every sentences we delete are in decreasing order
    def check_dec_order(self, seq):
        for i in range(1, len(seq)):
            if seq[i] >= seq[i-1]:
                print(seq)
                raise EnvironmentError("ERROR! order is not decreasing")

    ## use KB0 to update KB
    def update_KB(self):
        del_sen = []
        ## iterate every sentence in KB
        for sen_idx, sen in enumerate(self.KB):
            del_elem = []
            for elem_idx, element in enumerate(sen):
                if (element[1], element[2]) in self.KB0.keys():
                    if (self.KB0[(element[1], element[2])] and
element[0])\
                        or (not self.KB0[(element[1], element[2])] and
not element[0]):
                        del_sen.append(sen_idx)
```

```
                    break
                else:
                    del_elem.append(elem_idx)

        ## delete element which has already exist in KB0
        if del_elem != []:
            del_elem.reverse()
            self.check_dec_order(del_elem)
            for idx in del_elem:
                del self.KB[sen_idx][idx]

    ## delete sentence which has no longer needed since it always
true
    if del_sen != []:
        del_sen.reverse()
        self.check_dec_order(del_sen)
        for idx in del_sen:
            del self.KB[idx]

## add global constraint
def add_global_constraint(self):
    tot_mines, tot_cells = self.mines_num, self.x_size*self.y_size
    true_elements, false_elements = [], []

    ## iterate every cell which hasn't exist KB0
    for y_idx in range(self.y_size):
        for x_idx in range(self.x_size):
            if (x_idx, y_idx) in self.KB0.keys():
                tot_cells -= 1
                if self.KB0[(x_idx, y_idx)]:
                    tot_mines -= 1
            else:
                true_elements.append([True, x_idx, y_idx])
                false_elements.append([False, x_idx, y_idx])

    ## if there're too many empty cells, break it
    if tot_cells >= 10:
        return
```

```
        pos_literals = list(itertools.combinations(true_elements,
tot_cells-tot_mines+1))
        neg_literals = list(itertools.combinations(false_elements,
tot_mines+1))

        ## add new sentence to KB
        for sen in pos_literals:
            self.insert_KB(list(sen))
        for sen in neg_literals:
            self.insert_KB(list(sen))

    ## process the board
    def process(self, Env):
        prev_KB_len, prev_KB0_len, cnt = -1, -1, 0
        with_global_constraint = False
        success = True
        while len(self.KB):
            ## if the length of KB0 and KB hasn't change for continuous
5 times
            ## add global constraint, or break it
            if prev_KB0_len == len(self.KB0) and prev_KB_len ==
len(self.KB):
                cnt += 1
                if cnt > 5:
                    if not with_global_constraint:
                        cnt = 0
                        with_global_constraint = True
                        self.add_global_constraint()
                    else:
                        success = False
                        break
            else:
                prev_KB0_len = len(self.KB0)
                prev_KB_len = len(self.KB)

            ## sort the KB by the number of length of sentence in
increasing order
            self.KB = sorted(self.KB, key=lambda x: len(x))

            ## if the length of first sentence is only 1, expand it
```

```
            if len(self.KB[0]) == 1:
                flag, x, y = self.KB[0][0][0], self.KB[0][0][1],
self.KB[0][0][2]
                self.KB0[(x, y)] = flag
                self.board[y][x][0] = True

                if not flag: ## if it is not mine
                    mines_cnt = Env.get_sur_mines_num(x, y)
                    self.board[y][x][1] = mines_cnt
                    self.gen_clause(x, y, mines_cnt)
                else:
                    self.board[y][x][1] = -1

                del self.KB[0]  ## delete this sentence
            else:
                ## if the length of sentences are all greater of equal
to 2, matching it
                self.matching()

            ## update KB since we have new KB0
            self.update_KB()

        return success
```

```
import obj
import time

def experiment_one(x=30, y=16, safe_cells_num=22, Times=100):
   res_cnt = [0, 0]
   print(f'x_size {x}, y_size {y}, test times {Times}')
   for mines_num in range(50, 100, 5):
       res_cnt = [0, 0]
       tot_time = 0.0
       for _ in range(Times):
           Env = obj.Environment(x, y, mines_num)
           safe_cell_list = Env.give_safe_cells(safe_cells_num)

           start = time.time()
           Player = obj.Agent(x, y, mines_num, safe_cell_list)
           res = Player.process(Env)
           this_time = time.time() - start
           tot_time += this_time
           if res:
               res_cnt[0] += 1
           else:
               res_cnt[1] += 1
       print(f'{mines_num}: success {res_cnt[0]}, lose {res_cnt[1]},
time {tot_time/Times}')

def experiment_two(x=9, y=9, mines_num=10, start_safe_num=1,
Times=100):
   print(f'x_size {x}, y_size {y}, mines_num {mines_num}, test times
{Times}')
   for safe_cells_num in range(start_safe_num, x*y, 5):
       acc_cnt = 0
       tot_time = 0.0
       for i in range(Times):
           Env = obj.Environment(x, y, mines_num)
           safe_cell_list = Env.give_safe_cells(safe_cells_num)

           start = time.time()
           Player = obj.Agent(x, y, mines_num, safe_cell_list)
           res = Player.process(Env)
           this_time = time.time() - start
```

```
            tot_time += this_time
            if res:
                acc_cnt += 1
        accuracy = acc_cnt/(Times)
        print(f'{safe_cells_num}: accuracy {accuracy}, time
{tot_time/Times}')
        if accuracy >= 0.9:
            break


# experiment_one()
experiment_two(x=30, y=16, mines_num=99, start_safe_num=200, Times=20)
```