Minesweeper AI with expected value

孫宜君(33%) 109550118
陳宥安(33%) 109550073
楊竺耘(33%) 109550139

Github repo: https://github.com/yachen0409/AI_final
Demo video: https://youtu.be/Gzj6q7J6lxc

1. Introduction

本次期末專案，我們將以期望值代入踩地雷遊戲所成的演算法做討論。踩地雷是一款家喻戶曉的遊戲，其規則簡單，深受世界各地人們的喜愛，但這項看似簡單的遊戲，玩起來卻不是那麼容易，很難做出最完美的路徑嘗試，但我們仍然不斷運用各種方法，試圖達到最好的結果。

由於我們三人對此遊戲都有一定程度的熱愛，僅僅面對不同隨機情況的挑戰並不能滿足我們對此遊戲深入了解改善的慾望，若能透過更精確的方式預測安全的路徑，並創造更好的踩地雷模型，不但能確實應用這學期在人工智慧概論這門課所習得的知識，還能從這次實行中獲得偌大的成就感。

Straightforward Algorithm是大多數踩地雷遊戲最常用來預測安全區域的演算法，這次project的目的就是將Straightforward Algorithm中沒有百分之百確定為安全沒地雷的區塊時，採用隨機挑選的方式更改為計算期望值判斷，增加挑選正確的機率，並盡量成功贏得遊戲。除了基本的預測一步外，也增加讓AI直接從頭至尾跑完一整局遊戲的功能。

2. Related Work

踩地雷遊戲主要由四個元素組成。「方格」為未知的資訊，只有按下按鈕玩家才可得知;「空白」為附近八格都沒有炸彈的安全位置;「數字」為八方的地雷數，範圍是一到八的正整數;當按下「地雷」時遊戲結束 [1] 。關於踩地雷的AI，大多採用Straightforward Algorithm，只追求盡可能快速得到比賽的勝利，不考慮過程中輸掉多少比賽的代價 [2] ，過程中以唯一解尋找沒有地雷的位置，也就是只利用兩種情況判斷:方塊數加旗標數等於數字值、旗標數等於數字值，若兩者都不符合則完全隨機選擇。

3. Methodology

統整我們的想法，繪製出程式運作的流程圖:

Figure 1

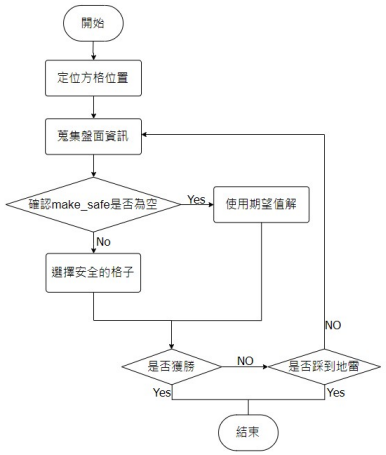

Figure 2

```
for neighbor in self.knowledge:
    for element in neighbor.cells:
        # print("element:",element)
        if element not in cell_freq:
            cell_freq[element]=1
        else:
            cell_freq[element]+=1
        for i in range(element[0] - 1, element[0] + 2):
            for j in range(element[1] - 1, element[1] + 2):
                if 0 <= i < self.height and 0 <= j < self.width:
                    if (i,j) in self.moves_made:
                        neighbor_num=0
                        for row in range(i-1,i+2):
                            for col in range(j-1,j+2):
                                if 0 <= row < self.height and 0 <= col < self.width:
                                    if (row, col) not in self.moves_made and (row, col) not in self.mines:
                                        neighbor_num+=1
                        if element in Dict:
                            Dict[element]+=self.countDict[(i,j)]/neighbor_num
                        else:
                            Dict[element]=self.countDict[(i,j)]/neighbor_num
                        if element in self.mines:
                            del Dict[element]
for index in Dict:
    Dict[index]=Dict[index]/cell_freq[index]
self.jimmy=Dict
```

上圖是用來計算各點的期望值，首先遍尋周圍的點，並且將這次遍尋的點的出現頻率紀錄在cell-freq中，接著再將這些點的周圍也遍尋一次，計算出周圍沒有走過的點的數量，紀錄於neighbor-num;將該點的周圍地雷數除以neighbor-num，並且採持續累加，最後再除以該點出現的頻率，如此便計算出該點的期望值。

Figure 3

```
available_steps = self.safes - self.moves_made
if available_steps:
    return random.choice(tuple(available_steps))
elif self.jimmy:
    temp_jimmy = sorted(self.jimmy.items(), key=lambda item: item[1])
    return temp_jimmy[0][0]
return None
```

將百分之百確定沒有地雷且未經過的位置放進available-step中，若是available-step不為空，就從此選出下一步，若為空，則從存放由掉大牌續各點期望值的jimmy中取最小期望值的點作為下一步。

Figure 4

```
if test and testCount!=10000:
    if game.mines==flags:
        scoreBoard.add(len(ai.safes)*2)
        testCount+=1
        win_num+=1
        autoplay=True
        game = Minesweeper(height=HEIGHT, width=WIDTH, mines=MINES)
        ai = MinesweeperAI(height=HEIGHT, width=WIDTH)
        revealed = set()
        flags = set()
        lost = False
        if testCount % 100 == 0:
            print("No.", testCount," iteration-----------------")
    elif lost:
        # if len(ai.safes) >= 10:
        #     lostBoard.add(len(ai.safes))
        # # else:
        # #     lostBoard.add(1)
        if len(ai.safes) >= 1:
            scoreBoard.add(len(ai.safes))
            lostBoard.add(len(ai.safes))
            bigger_lost_num += 1

        testCount+=1
        lost_num+=1
        autoplay=True
        game = Minesweeper(height=HEIGHT, width=WIDTH, mines=MINES)
        ai = MinesweeperAI(height=HEIGHT, width=WIDTH)
        revealed = set()
        flags = set()
        lost = False
        if testCount % 100 == 0:
            print("No.", testCount," iteration-----------------")
```

上圖為計算勝負和勝率的方法，分別計算獲勝的場次所走的步數，再乘以二，和輸的場次所走的步數相加，即為分數的算法，接著利用lost-num計算輸的場次，以便計算勝率。

Figure 5

```
if autoplay:
    move = ai.make_safe_move()
    if move is None:
        move = ai.make_random_move()
        if move is None:
            flags = ai.mines.copy()
            # print("No moves left to make.")
            autoplay = False
        else:
            random+=1
            # print("No known safe moves, AI making random move.")
    # else:
    #     print("AI making safe move.")

    # Add delay for autoplay
    if autoplay:
        time.sleep(autoplaySpeed)
```

若是按下autoplay按鍵，則進入上圖執行，透過make-safe-move計算出下一步，若是沒有百分之百安全的區域，則進入計算期望值的步驟，若是踩到地雷，則立即停止autoplay。在過程中一併計算使用random選擇下一步的次數。

4. Experiments

為了做比較。我們將我們用Expected value的方法和網路上用Straightforward的方法各跑10000次遊戲，來計算各自的勝率（完全破關）、平均步數（輸了的情況下）、平均分數和平均分數（輸了的情況下）。而我們對勝利的定義為「走完整張圖上所有可以走的點並且沒有踩到地雷」，分數的計算為走了幾步就得幾分。如果勝利的話分數會乘以2。

根據以上規則，以下圖二圖三是我們的實驗結果：

Figure 6. Expected value

```
No. 10000  iteration-----------------
------------------------------------------------------------
Won rate= 0.4589
Average step(if lost)= 4.281422924901186
total random step= 20246
average score= 109.01960784313725
average score when loss= 106.89473684210526
------------------------------------------------------------

AI_final on  master (MERGING) [=!⇕] via 🐍 v3.8.10 took 48m41s
❯
```

Figure 7. Straightforward algorithm

```
No. 10000  iteration-----------------
------------------------------------------------------------
Won rate= 0.4514
Average step(if lost)= 4.104092976250632
total random step= 30896
average score= 106.8974358974359
average score when loss= 104.8
------------------------------------------------------------

AI-Minesweeper on  main [!] via 🐍 v3.8.10 took 39m52s
❯
```

由測試結果我們得知，用期望值來作為minesweeper AI決定下一步的方法的確可以帶來較高的勝率，但差異並沒有跟Straightfoward差到想像中的那麼大。

我們推測其中的原因可能是因為:

- 資訊量不足
  當遊戲一開始藉由期望值來做下一步的決策時，在訊息量不夠多的情況下是會有很多點期望值都是相同的，而這時我們只能隨機從最低期望值取一點出來，從另一個角度來看，也是跟Straightforward 中當沒有安全可走的step時，要隨機走一點是差不多的意思。

- 起始點問題
  不論是Straightforward又或是expected value，第1點都是透過random下隨機挑選一個點的，跟人來玩踩地雷一樣。而對踩地雷來講，第1步是很重要的，但不論如何第1步都是靠運氣，再加上每一輪測試地雷位置都是隨機生成的，因此無法保證每一輪都有一個好的起點。

References

[1] https://www.shs.edu.tw/works/essay/2020/09/2020091022104133.

[2] https://luckytoilet.wordpress.com/2012/12/23/2125/.