人工智慧期末專題書面報告

2048

B99203027 陳柏豪
R03921084 傅冠鈞
R03921089 肖可依

一、摘要

在此 project 中，使用了Python 作為實作時的語言。實作的關鍵在於評估函式 (evaluation function) 與搜尋演算法 (search algorithm)。我們實作了數種評估函式，包含蛇行 (snake)、空白格 (empty tiles)、單調性 (monotonicity) 與平滑性 (smoothness)。搜尋方法為一種類似 expectiminimax 的方法，並在最後加入了適應性調整深度方法 (adaptive depth method) 與機率方法 (probability baced method)。
實驗結果顯示，使用蛇形評估函式與適應性調整深度方法有 90% 以上的機會可以達到最大方塊數字 2048 以上，並有約為 8% 的機率可以達到 8192。
使用機率方法可以顯著地加速，並在大部分狀況下得已達到 4096。

二、簡介

2048 是一名義大利人 Gabriele Cirulli 於 2014 年所開發的一款遊戲。遊戲任務是在一個四乘四的盤面執行上、下、左、右等動作，直到達成最大數字方塊數值為 2048。當兩個相同數字的方塊聚集在一起則合成一個此數字兩倍的方塊。是一個數學邏輯結合方塊移動的遊戲。一般人的程度可以玩到 1024，亦不少人在一定的時間內玩到 2048，但達成 4096 的人便是少得多了，8192只有極少數的人達成，而達成 16384 的是極少極少。

我們採用可取得的 UI 界面，支持鍵盤輸入以移動方塊。我們加入了電腦自動算出最佳策略並自己移動方塊的模式。

可選取輸入的方式 (a / d / w / s / d / n / r / q)
輸入 ”a”、“d”、”w”、”s” 分別代表將方塊向 “left”、”right”、”up”、”down” 整排移動。
輸入 “n” 能夠看到電腦以搜尋演算法與評估函式決策出的下一步。
輸入 “r” 電腦會直接一路玩到無路可走，即最後盤面。
輸入 “q” 離開遊戲。

其中一些程式碼 (operation functions) 敘述如下：

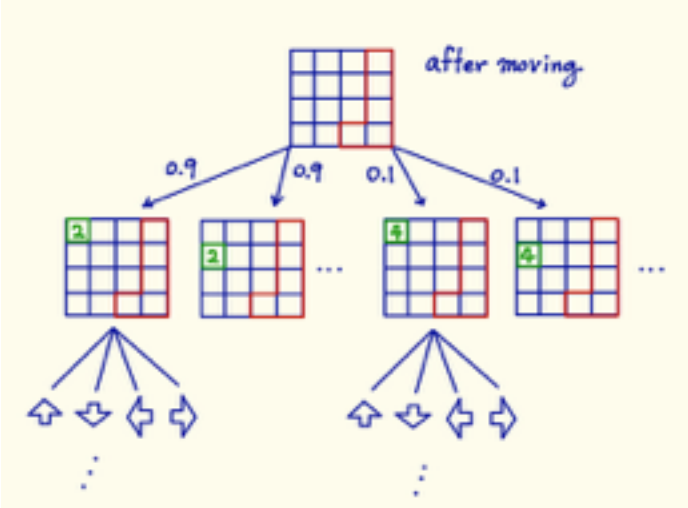
Get_next_action(): up (向上移)，down (向下移) ，right (向右移)，left (向左移)。
Push_row(): 執行 right 或 left 時，把一個橫列聚集在一列，將會把相同數字合成其數字兩倍。
Get_column(): 回傳一個 list，此 list 儲存全部直行。
Set_column(): 把相對應的值儲存到各別的 list 直行中。
Push_all_rows(): 把全部 list 的橫列執行 Push_row() 指令。
Push_all_columns(): 先藉由 Get_column 指令，拿到全部 list 的直列，再利用 Push_row() 指令把全部 list 的直列聚集在一起，最後執行 Set_column() 把全部的值儲存到個別的 list 直行中。

Any_possible_moves(): 回傳一個 boolean 值，若有 legal move 則回傳真，否則回傳假。
Get_start_grid(): 回傳一個 list，此 list 儲存一個新的遊戲盤面 (gird) 狀態。
Prepare_next_turn(): 回傳一個 list，此 list 儲存執行下一步所產生的遊戲盤面 (grid) 狀態。
Print_grid(): 印出遊戲盤面 (grid) 的當前狀況。

三、內容

1. Implementing expectiminimax()

我們一開始採用類似於 expectiminimax 的方案，但稍作了點修改。一般的 expectiminimax 會有個 min 層，是遊戲本身有意走出特定步數以減低盤面的分數。但在此遊戲中，“2” 和 “4” 的出現是純然隨機的，因此並沒有 min 的成分。我們將原本的 expectiminimax recurrence 改寫為：當輪到電腦時，電腦會產生所有可能的情形，即新的數字出現在當前空格的所有可能。以這些新展開的盤面為基準，在更深的一層假設四種更動盤面的可能性 (e.g. up, down, right, left) 以更動後的盤面為基準再繼續遞迴下去。在末端節點處選擇盤面分數最大的一支，其路徑上的第一個更動方向作為真實遊戲中的下一個決策。
此方法如下圖所示：



紅色部分代表已有數字的區域，其餘為空格方塊，在 0.9 與 0.1 的機率下 (產生 ”2” 或 ”4”) 分別考慮所有新方塊的可能性，即綠色方塊，並繼續遞迴地考慮四種移動方式所產生出來的分支。

```
def expectiminimax(grid, player, depth):

    # check terminal condition
    if depth == 0:
        return evaluation_function(grid)

    # computer's turn (consider all random conditions)
    if player == 0:
        alpha = 0

        for x, y in get_empty_cells(grid):
            grid[x][y] = 2
            alpha = alpha + prob_2 * (1 / number_of_empty_cells) * expectiminimax(grid, 1, depth - 1)

        for x, y in get_empty_cells(grid):
            grid[x][y] = 4
            alpha = alpha + prob_4 * (1 / number_of_empty_cells) * expectiminimax(grid, 1, depth - 1)

    # AI's turn
    elif player == 1:
        alpha = 0

        for move in ['up', 'down', 'left', 'right']:
            functions[move](grid)
            alpha = max(alpha, expectiminimax(grid, 0, depth - 1))

    return alpha
```

程式部分亦可看到實作被分為三個部分，check terminal condition、computer's turn 以及 AI's turn。

此方法本質上有個嚴重的缺陷：
搜尋深度僅能推至 4 左右 (若以電腦與 AI 各走一次作為一輪深度的話實際深度應為 2)，因為考慮所有可能出現在當前空格的可能性不少，若要遞迴地往下展開更多節點必然產生指數性暴增。我們因而採取了適應性調整深度方法 (adaptive depth method)，即於不同盤面狀況以不同的深度搜尋。判斷的依據是當前盤面的空格數，若空格數多則搜尋深度淺 (不必想太多)，若當前空格數少，則搜尋深度深 (盤面擁擠時必須多想)。如此便可在合理的時間內達到不錯的效果。

由實驗得到的結論是，若能將深度推得很深，即使簡單的評估函式亦能有非常好的效果。因此加深深度成為我們後來改進的主要標的。

2. Evaluation Function

(1) emptyTiles()

最初的嘗試是使用非常單純的評估函式，僅考慮當前盤面的空格數作為其評分。雖然單純但效果意外地好。絕大部份時候能達成最大分數為 512，運氣好的時候能跑到 1024。使用 emptyTiles() 的效果可以很容易地在執行時看出，所有的方塊都會盡可能地聚集合併，以騰出更空格。但這樣的移動缺乏策略，許多數字大的方塊散落在各處，等盤面擁擠後便不容易合併，加上搜尋深度有限，無法規劃出能將大數字方塊合併的決策，因此有必要加入其他評估策略。

(2) emptyTiles() + smoothness()

為了使數字較大的方塊不要分離得太開，以致盤面複雜時難以合併，我們加入了 smoothness() 作為輔助 emptyTiles() 的評估函式。此函式計算了每個方塊與自身周遭的方塊數字差值。遍歷所以數字方塊後將此分數加總後成為此函式的評分。這個分數越高代表盤面狀況越複雜，即大小數字交錯且數值相差大，不利於合併，故應避免。將總和的值倒數即可表示平滑性，並與 emptyTiles() 的評分以適當的權重線性相加作為一個盤面最後的評分。使用這個組合可使大部分情況的最大方塊數字落在 1024，甚至有時候可以達到 2048。

(3) emptyTiles() + smoothness() + monotonicity()

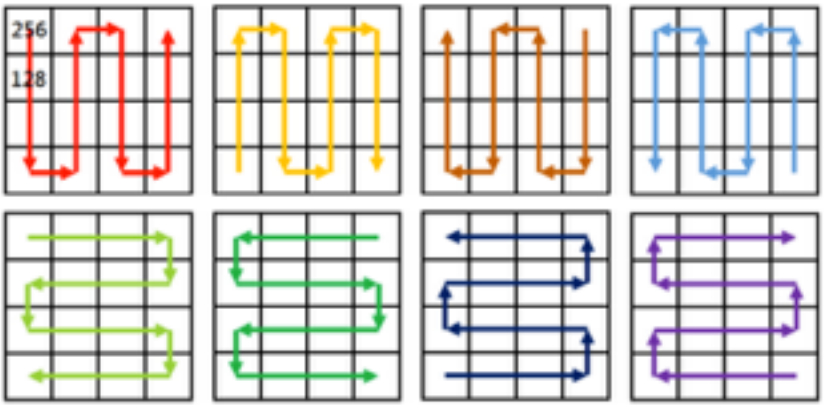
這部分又多加入了一個評估函式，目標在使整體的方塊數字分佈呈現出單調性。這樣一來大分數的方塊會傾向於以同一種方向遞增或遞減排列，使得數字呈現等比的可能性變高，利於合併。但實作時發現這個函式給出的評分變動很大，經過線性縮放後仍難以加入原評估函式。實際選用了一個線性組合測試執行，表現並沒有顯著提升，因此最後簡報時並未呈現這個評估函式。

(4) snake()

若使用更精確的盤面描述，便可在有限的搜尋深度達到足夠好的效果。我們實作了一種以等比數列排列為概念主軸的方法。實際公示如下方所示：

$$\sum \text{value}(\text{tile}(n)) * (r^n)$$

式子中 n 為 0 ~ 15，tile(n) 為第 n 個方塊，value() 則取出其中的數值，r 為一自定的參數 (根據經驗定 r = 0.5)。從此式可看出盤面會傾向於將數字排列成等比數列，如下圖所示：



在 n = 0 的方塊受到的衰減最小，n 越大的方塊則重要性越低。n 增加的順序即為圖中所示的順序。評估盤面時若有一種蛇形排序強於另一種便會立刻改變排序的方案，例如從左上角的紅色蛇形變為深綠色蛇形。這點便是人類較難想到的，人類習慣將大的數字卡在固定角落，而較少在適當的時機轉換安排盤面的方式，以創造未來更多的彈性與選擇。這個評估函式的效果非常好，因為它的策略幾乎就是此遊戲的破解關鍵——將數字排成一串等比數列。在深度為一層時便有不錯的效果。加上適應性調整深度方法，深度設定為空格多時 depth = 2 (computer 走一輪，AI 走一輪，實際深度是一層，程式中 depth = 2)，空格少時 depth = 4。如此一來便可有非常好的結果，約有四分之一的最高分數方塊落在 2048，而 4096 占了四分之一，有約莫十分之一的可能走出 8192。

3. Pruning

由於實作 expectiminimax() 的方式使得搜尋深度過淺，必須倚仗著強大的評估函式才能走出不錯的結果，因此加速或是改變演算法成為改善的目標，以增加搜尋深度。我們嘗試了對搜尋過程進行剪枝，也就是不考慮所有可能的盤面狀況，而在不失真的情況下，將某些意義上相等的盤面狀況統整起來。例如新出現在同一列的數字方塊，往左或右都會被推到最底，因此會有一樣的盤面分數。原做法將出現在同一列數字方塊的所有可能性都當做個別狀況繼續向下展開，因此分支的數量會瞬間大到無法承受。

```
def prunedExpectedminimax(grid, player, depth, nextMove):
    height = len(grid)
    width = len(grid[0])

    if depth == 0:
        return score_evaluation_snake(grid)

    # computer's turn (randomly)
    if player == 0:
        alpha = {'a': 0, 'd': 0, 'w': 0, 's': 0}
        numEmptyCells = len(get_empty_cells(grid))
        #up/down
        for j in range(width):
            i = 0
            while i < height:
                count = 0
                while (i < height and grid[i][j] == ''):
                    i = i + 1
                    count = count + 1
                if (i > 0 and grid[i - 1][j] == ''):
                    grid[i - 1][j] = 2
                    alpha['w'] = alpha['w'] + prob_2 * count / numEmptyCells * prunedExpectedminimax(grid, 1, depth -1, 'w')
                    alpha['s'] = alpha['s'] + prob_2 * count / numEmptyCells * prunedExpectedminimax(grid, 1, depth -1, 's')
                    grid[i - 1][j] = 4
                    alpha['w'] = alpha['w'] + prob_4 * count / numEmptyCells * prunedExpectedminimax(grid, 1, depth -1, 'w')
                    alpha['s'] = alpha['s'] + prob_4 * count / numEmptyCells * prunedExpectedminimax(grid, 1, depth -1, 's')
                    grid[i - 1][j] = ''
                while (i < height and grid[i][j] != ''):
                    i = i + 1
```

新的做法希望能經由統整，使得分支數量顯著變少。這個方法的嘗試最後以失敗告終。因為這個統整方法的本質必須在新的方格出現以前先考量下一個步驟欲採取的移動方向，但由於 expectiminimax() 的實作，出現在不同格子的新數字方塊是在展開搜尋樹同層的不同分支，而考量移動方向是出現方塊後的下一步。用了許多方式改寫 expectiminimax() 都無法在不失真的情形下得到好的結果，因此剪枝的方案最後並沒有作為主要的。

4. Probability Baced Method

最後我們嘗試了以機率為主軸的加速方法。這個方法捨棄了原本 expectiminimax() 的寫法，不再考慮所有可能的出現的方格，改變為隨意考慮某個出現的方格，再以此盤面為基準，考慮四種方向繼續往下延伸。設定往某方向最為初始移動策略的循環次數，便可分攤掉出現較差盤面狀況的機率。設定循環次數方法為：

loop = 2 * (number of empty cells) ** depth

```
def probabilityMethod(grid,depth):
    grid_copy = copy.deepcopy(grid)
    num_empty_cells = len(get_empty_cells(grid))
    loop = 2 * (num_empty_cells) ** depth
    total_score = 0

    for i in range(loop):
        grid = copy.deepcopy(grid_copy)
        for j in range(depth):
            max_score = 0
            best_move = ''
            for move in 'adws':
                grid_temp = copy.deepcopy(grid)
                functions[move](grid_temp)
                if grid_temp != grid:
                    score = score_evaluation(grid_temp)
                    if score > max_score:
                        max_score = score
                        best_move = move
            functions[best_move](grid)
        total_score = total_score + max_score
    return total_score/loop
```

由此循環次數的設定可看出，在剩下空格較多的情況下，重複隨機的次數便會提高，而其指數取決於深度。意即用趨近於原本 expectiminimax 分枝的循環次數去平攤較差狀況的機率。但此方法無論如何都會快於 expectiminimax，因為在許多情形下，循環次數是小於考慮所有狀況的分枝數的，而當情是不複雜時，約略的失真是可以承受的。最後的結果在深度以適應性調整為 depth = 1, 2, 3 的情形下，約有 15% 走到 1024，36% 走到 2048，並且有 47% 走到 4096。

以機率方法進行搜索，可在合理的時間內加深搜尋深度至至多四層，即 expectiminimax 方法中的八層。而直接使用 expectiminimax 僅能搜尋至六層 (即機率方法中的三層)。

四、結論

原本使用了 emptyTiles() 與另外兩個補充的評估函數能達到的成效有限，換為 snake() 後有顯著改善。再加上適應性調整深度方法可以達到不錯的成果。中途嘗試使用剪枝，但成效不彰。最後使用了基於機率的搜索方法，顯著地改善了速度，並且在時間允許的情況下可無限制地增加搜索深度 (解決的空間複雜度的限制)。

另外的加速方案還有將盤面編碼。由於我們的實作是將盤面存為一個矩陣，每次複製以及改動都會用到不少運算資源，因此若將盤面編為二進位以搬動位元的方式將運算放到較底層，就有機會加速運算。但這個方案本身與人工智慧語言算法較不相干，是一種純工程上的改進，因此在參考了別人的加速成果後並沒有積極地朝這個方向改進。

附註：後來一查之下發現 2048 是一款抄襲 Threes! 的遊戲，並在手機遊戲界鬧得沸沸揚揚，而 2048 作者亦出面承認抄襲並道歉。做這個 project 前並未查知這樣的事實，否則可能會更傾向於以原作的遊戲為題。

五、參考書目

[1] 葉駷豪、梁朝欽、吳毅成，「2048人工智慧程式」

[2] stackoverflow, “What is the optimal algorithm for the game, 2048?”

[3] Nicola Pezzotti, Diary of a Tinker, “An Artificial Intelligence for the 2048 game”