109550175 許登豪 AI HW2 Report

1.Homework overview

在這次作業裡我實作了兩種演算法 : 分別是 MCTS 以及 Minmax with alpha-beta pruning . 這兩種演算法我都 fine tune 參數使得他們可以在限制的 6 秒內做出合理的 move。接下來的報告會包括以下幾個部分 :

(1) 介紹這兩個演算法如何實作
(2) 對於各種算法做交叉比對並且給出為何會有如此現象的推論
(3) 我的體驗以及未來可以嘗試的作法
(4) 附錄，links 內含實作的 code 以作證明。

2.Introduce how to implement these 2 algorithm using Python

首先介紹 Minmax with alpha-beta pruning . 先給出 alpha beta pruning 的 psuedo code :

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if value ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if value ≤ α then
                break (* α cutoff *)
        return value
```

```
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Source : https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

這裡介紹的是 fail-soft 的版本，也是我所實作+比賽時選用的版本(深度為 6，當盤面剩下 11 個 node 的時候才會啟用 Minmax algorthm，否則隨機做動作) : 而 fail-hard 的版本則是把更新 alpha(beta)值和 beta(alpha) cutoff 這兩者的順序做了交換。這個代表的涵義在於，fail-hard pruning 是一個更激進的策略: 只要滿足了當前的 cutoff condition 則會停止搜索。fail-soft 則會先進行更新 alpha-beta 值再做 cutoff，所以相對來說可以探尋到更多的分支。

Alpha-beta pruning 隱含的意涵很簡單 ： 在遞迴這棵 GameTree 時如果我是 MaxPlayer(MinPlayer)並且拿到 x 分，那麼在我探索完這顆子樹所有可能動作後我必會選擇 >= x 分(<= x 分)的動作，但當這個 x 分比 MinPlayer(MaxPlayer)目前探尋到的最佳分數--也就是他會想要給 MaxPlayer(MinPlayer)的最低(最高)分數還要高(低)時，MinPlayer(MaxPlayer)肯定不會給對家得逞(對家選不了)，因此可以省去探索這個子樹的成本。實作的 code 和 上方給的虛擬碼類似，唯一要注意的是遞迴的終止條件需要深思，這會在之後介紹 scoring function 中提到。

而對於 Minmax algorithm，我們需要定義一個 scoring 機制，也就是說我們需要對當前遊戲狀態有著一定的認知才行，相反的 MCTS 這種作法則不需要對原遊戲有任何理解，單純以足夠的模擬數作為動作選擇合理性的 backup。
以下介紹我的 Scoring 機制(分數為以下幾項做總和，權重都相等) :

(1) 必贏或必輸:輪到某方下棋時如果只剩一個 node , 則那一方必輸；如果輪到某方時沒得下則代表對家幫他把最後一步下完，他必勝。

```python
def DoomWinorLose(after_mapStat):
    num_zeros = np.count_nonzero(after_mapStat == 0)
    if num_zeros == 1:
        return -10000
    elif num_zeros == 0:
        return 10000
    else:
        return 0
```

(2) Lonely Node , 判讀周圍沒有鄰居的 Node 的個數之奇偶，如果盤面”僅存”Lonely Node 在場上，則奇數對現在下棋方必敗，偶數必勝；

(3) Pair Node , 判讀兩兩為一組的狀況有幾組，如果當前盤面”僅存”恰好兩組這樣的 pair 則當前現在下棋方必敗。

```python
def ScatteredNode(after_mapStat):
    after_zero_coords = checkRemainMove(after_mapStat)
    lonely_node_count = 0
    pair_node_count = 0
    for coord in after_zero_coords:
        x , y = coord[0] , coord[1]
        neighbor = 0
        for dir in [1 , 2 , 3 , 4 , 5 , 6]:
            neigh_x , neigh_y = Next_Node(x , y , dir)
            if neigh_x < 0 or neigh_x > 11 or neigh_y < 0 or neigh_y > 11:
                continue
            if after_mapStat[neigh_x][neigh_y] == 0:
                neighbor += 1
                if neighbor > 1 :
                    break

        if neighbor == 0 : lonely_node_count += 1
        if neighbor == 1 : pair_node_count += 1


    if len(after_zero_coords) != (lonely_node_count + pair_node_count) :
        # assert board contains only OO or O like segments
        return 0

    assert (pair_node_count%2) == 0
```

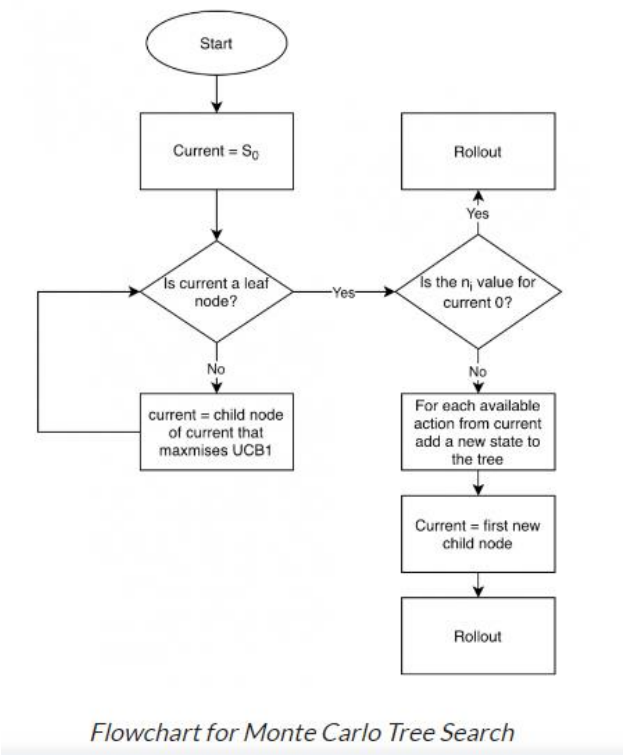
```python
    if pair_node_count == 0 : # which means only lonely nodes are on the board
        if (lonely_node_count%2) == 1:
            return -9000
        else:
            return 9000
    elif lonely_node_count == 0 : # which means only OO are on the board
        if (pair_node_count/2) == 2:
            return -9000
    else :
        pass

    return 0
```

其中這兩個情況都只考慮場上單純只有 Lonely Node 或是 Pair Node 的狀況，若有混合的狀況則期望藉由多次遞迴模擬收束到上述幾種狀況。反之則 return 0，和其他狀況下任意盤面得分一致，這也隱含著不刻意創造上述幾種情況的保守策略。

這個 scoring function 我想了一陣子，因為他要足夠強大到可以直接判定勝負，又要足夠簡單+能快速計算到可以放在 6 層的 Minmax 中。
要注意 Minmax 的 terminate state 要寫成以分數做判斷，如果以盤面上有沒有 0 做判斷將會導致必勝的局被略過(被視為非終止條件而進行下一層遞迴模擬)而在導致必敗的局中做選擇。同時因為我的 scoring function 是不管 min max player 的，所以我在遞迴回傳分數值時需要調整對應正負號。(請看下圖)

```
def best_action(self , is_max_player , cur_mapStat , alpha , beta , depth):
    abs_score = Total_Score(cur_mapStat)
    if (depth == self.max_depth) or abs(abs_score) >= 10000 :
        signed_score = abs_score if is_max_player == 1 else -abs_score
        return None , signed_score
```

接下來介紹 MCTS，先給上 MCTS 的 flow chart :



Flowchart for Monte Carlo Tree Search

Source : https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/
MCTS 期望以一個足夠大的模擬次數當作基底，計算各動作的表現並且從中選擇。其中在每一層往下選擇子代的方式為計算 UCB value，終止條件為選擇到某個葉節點，此時若這個葉節點被拜訪過則我們會根據這個葉節點所有可以嘗試的動作建立對應子節點，再隨機選擇一個子節點進行 rollout(若葉節點未被拜訪過則可以直接 rollout)。Rollout 意味著由當前 node 狀態一直模擬到遊戲結束，最後我們需要把這次勝負結果一路往上更新(backpropogate)，這時雙方玩家如何行動取決於如何定義 rollout policy，可以是隨機、Minmax based 或是經由

access 一個 neural network 得到動作等等。在此次作業中因為有 6 秒的限制，我採用隨機選擇的策略。

最後 root(也就是要拿到當前盤面 best action 的那個初始狀態)該選擇哪個動作的方式多採用計算對應動作的子樹的勝率，實作中我也是採用如此的策略。實作中唯一要注意的是在計算子樹 UCB values 時，要根據當前節點代表 player 1 還是 player 2 將對應勝場數放到 UCB 公式中的 w(見下圖)，這點在很多網站上都沒有詳細解釋，足以讓人誤解。

```
def UCB(self , c_param): # calculate the UCB value of "self"
    if self.number_of_visits == 0:
        return 1000000
    w = None
    n = self.number_of_visits
    if self.turn == 1: # if "self" is the turn of player1 , since we only store win time of player1
        w = self.win
    else : # n - self.win == player2 win time
        w = n - self.win

    parent_simulation = self.parent.number_of_visits
    return w / n + c_param * np.sqrt(np.log(parent_simulation) / n)
```

可以看到過程中不用定義任何 scoring function，這是一個非常重要且強大的性質，方便我們在探索未知環境時使用而不需要任何 domain knowledge.

3. Experiment (each step in each exp. can be done in 6 seconds)

A. Random Player v.s. pure MCTS Player (550 times , c_param = 1.4) v.s. Minmax Player (random + 剩 11 個空格時再做 6 levels Minmax)

Random v.s. MCTS : 13 wins / 87 wins

Random v.s. Minmax : 0 wins / 100 wins

MCTS v.s. Minmax : 2 wins / 98 wins

推論 ： 可以觀察到 MCTS 的勝率仍然被 Minmax 所輾壓，我認為這可能是因為

1. Minmax 的 score function 是和遊戲勝負直接相關，相反的 MCTS 以”一個足夠大的模擬次數作為統計母體去選擇動作”這個想法和勝敗並非直接掛鉤。
2. MCTS 的模擬次數遠遠不夠，也可以推論在有時間限制下 MCTS 可能不適用於某些太複雜的遊戲，只能作為未知環境(遊戲)的初步探尋方式之一。
3. MCTS 的 random rollout 在模擬次數不夠多的情況下，可能會導致 MCTS 算法低估對手表現，相反的 Minmax 設定雙方每次動作都是合理的、最佳化己方利益的，如此較能反映出遊戲的真實狀況。
4. 模擬次數少的 MCTS 需要搭配其他算法如 policy net 或 Minmax 的 rollout 方式提升表現。

B. random+MCTS(剩 10 個空格時再做 MCTS(same 550 times)) v.s. pure MCTS(550 times) , c_param are both 1.4

Hybrid v.s. MCTS : 45 wins / 55 wins

推論 ： 藉由觀察到兩者勝率相差無幾，可以得出這個遊戲的初期動作並不是很重要，只需要到剩下不多空格的時候再集中算力模擬即可。

C. MCTS(550 times + c_param = 1.4) v.s. MCTS(same 550 times + c_param = 0.14)

MCTS 0.14 v.s. MCTS 1.4 : 20 wins / 80 wins

推論 ： 可以看到 c_param 為 0.14 的強度明顯低於 c_param 為 1.4 的，推測原因在於 c_param 太小，使得 MCTS agent 選擇動作的判斷依據很高一部分來自自己在初期探索過的幾個動作，而沒有繼續擴張探索其他可能的動作組合藉以得到更好的分數，因而表現遠不如後者。

D. Minmax Player(scoring function considers only terminate case) v.s. Minmax Player(scoring function is complete) , both are triggered if map contains less than or equal to 11 empty nodes

Incomplete v.s. Complete : 7 wins / 93 wins

推論 ： 因為只留下最終輸贏作為 scoring 依據，並且兩者都是在盤面剩下 11 個空格時觸發，因此在遞迴模擬選擇動作時此 agent 相較於有 Complete scoring 機制的 agent 更難提前預見雙方勝負(或是對哪方有利)因此避開或選擇關鍵動作。也就是說，在相同層數及觸發條件之下，有 Complete Scoring 機制的可以倚靠 scoring function 的設計，保證大部分狀況下遞迴模擬到最深深度時就可以預知比賽結果(或是對哪方比較有利)，而 Incomplete 的則很有可能因某些分支還未模擬到終局結果，資訊有缺失因而選擇錯誤動作。

E. Minmax Player(strongest version) battle each other

Player1 v.s. Player2 : 50 wins / 50 wins

推論 ： 遊戲先後手對勝負影響不大，可呼應前面提到遊戲初期不重要的觀點。

5. My Experience

我認為這次實作的過程相當有趣之外，6 秒的限制不僅要求了我們在此次作業中以運行起來最有效率的方式撰寫演算法，同時也讓我們可以體會到在現實生活中，如運用於工商業的機械手臂乃至自駕車等對即時性有要求的 AI，我們不能一味不惜成本與時間代價的選用最複雜的算法，

最後，我認為我之後可以嘗試的有幾個部分 ：

1. 採用 RL，利用 Q-learning 或是 DQN 當作動作選擇的依據
2. 模仿 policy net 利用 Conv2d 層或是 Transformer 將當前 mapState 輸入後，預測選擇某個最佳動作後的 after_mapState(一個 seq2seq 的 task)(我有在這次作業中嘗試，但是發現都 train 不起來，因而無法把好的結果放進報告中)

6. Appendix(Both are python files , not .exe)

Pure MCTS (it is not submitted with report , so I recommend TA check it , compile it and play with it !):

https://drive.google.com/file/d/1i4nJd5V2NmfWevwo4RxbH14wduSH48LC/view?usp=sharing

Minmax with alpha beta pruning :

https://drive.google.com/file/d/1ZK7NV798_YUOwGF8ZY1g0TDGPvPtNWUq/view?usp=sharing