AI group9 report 立旻驚人
0616222 黃立銘 0716003 鄒年城 0611289 謝旻紜

Achievement

Design a game-playing agent to play Othello/Reversi on 8*8 square board.

Comparison

我們最後共使用3個 exe 檔去倆倆比賽，分別是助教給的sample1(by random), 我們寫的兩個 MCTS, flip the most，其中MCTS故名思義，實踐蒙地卡羅搜尋樹，而flip the most方法是每次都找可以翻動最多對手棋子的點去下。
三種對局個別執行60次（分配給組員3人一人20次）

MCTS vs Random

	佔領格數 (all avg)	佔領格數 (win avg)	佔領格數 (lose avg)	勝場數	勝率
MCTS	37.7(62.8%)	42.3	27	44	73%
Random	22.3(37.2%)	33	17.7	16	27%

MCTS vs flip the most

	佔領格數 (all avg)	佔領格數 (win avg)	佔領格數 (lose avg)	勝場數	勝率
MCTS	33.6(56%)	36	24	49	82%
Flip the most	26.4(44%)	36	24	11	18%

Flip the most vs Random

	佔領格數 (all avg)	佔領格數 (win avg)	佔領格數 (lose avg)	勝場數	勝率
Flip the most	30.8(51.3%)	37.6	24	33	55%
Random	29.2(48.7%)	36	22.4	27	45%

Observations

MCTS 的表現比我們想像中的還要來得好，在與其他兩種對局中，超過一半的機率會贏得比賽，尤其是在與random比較下，結束時的佔領超過60%的格子。

原以為會是flip most 應該會比random來得好，畢竟在下黑白棋的時候，會下意識去尋找可以翻最多旗子的位置去下，但由上面結果可以知道當flip most vs random時

，在執行60場的情況下，幾乎是平手的狀態，且棋局結束時的佔領數幾乎是一樣多的。

Interpretation

當輪到我們時，會call function GetStep 去執行下棋的動作Mystate 為global，用來記錄這次是我們所拿到的是黑棋or白棋。

```
mystate=((is_black+1)%2)+1 #isblack=1->mystate=1/ isblack=0-> mystate=2
```

利用 x, y= board_class.get_action() 去call 我們所寫的function 最後回傳一個move x, y 去給client 執行下棋的動作。

因為讀進來的board為二維，利用 function loction_to_move 將二維座標轉換成一維的參數move，在計算上比較方便，最後return前會再利用function move_to_location，將一維[0~63]轉換回二維座標[0~7][0~7]。

第一次讀board進來以後將目前狀態下，空格沒有放置棋子的位置記錄在self.remain[]，而我們可以走的位置利用function get_available()，如果是邊界，判斷是否可以下旗子（下了以後會造成附近的棋子翻轉），如果是中間6*6的部分，只要是空的就可以下，將這些位置記錄在self.availables = [ ]。之後每次選擇好要下的位置後，利用function update 將availabes做更新。

```
def get_action(self): # 回傳一個move
```

因爲每次上限是五秒，所以當超過時間時就跳出while loop, return move。

```
while time.time() - begin < self.calculation_time:
```

在get_action() 內會先將讀進來目前狀態的board 複製一份，然後去重複做MCTS的計算，最後從計算結果中選出一個最好的走法，return move給getstep。

```
def run_simulation(self, board, play_turn):
```

為執行MCTS計算的部分，MCTS的主要分成 Selection, Expand, Back-propagation這三個part。
在模擬的過程中，會紀錄被selected的路徑上的所有走法存在visited_states裡

1. Selection 會從availables裡選出UCB最大值，做為下一步
2. Expand 每一次都只模擬最多expand一層深度的狀況
3. Back-propagation 將這次模擬的狀況向前面傳播，把所選路徑上的祖先節點的被選擇的次數都+1，以及獲勝方的次數+1。

```
def can_flip(self, move): # 判斷下了棋之後 有哪些棋子要被翻轉
```

在挑選要下哪個位置時，利用function can_flip() 去判斷，當下這格時，周遭會有哪些位置的棋子要被翻轉，將周遭八個方向都走一遍，當某一個方向為敵方（因為不為空，也不是自己）就一直走下去直到找到自己，如果某方向都是敵方（找不到自己），代表這一個方向沒有任何的棋子會被翻轉。

Things we learned

1. minimax search

2. alpha-beta pruning

3. MCTS：這是我們所選擇比賽用的算法，因為在我們一開始選擇要以什麼算法來比賽的時候，在網路上有看到有人提到alpha-beta pruning的結果並不是很好，而那個人用MCTS實踐的效果很不錯。

4. UCB：這是MCTS裡用來選擇點的公式，公式如下，

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

其中w_i表示移動i次之後勝利的 次數；
n_i表示移動i次之後 simulation的次數；
c是一個常數，決定exploit和exploration的比例，c越高越著重於exploit，在我們的程式裡面我們的c=1.96；
而t是總共的simulation次數，也就是n_i的和；
加號前面那項是exploration的部分，後者則是exploit的部分。

Remaining questions

在瞭解了pdf上所提到的三個方法後，最後選擇用MCTS來實踐，所以不知道minimax和alpha-beta會不會比較好，因為我們沒實踐這兩個。

Problem we met

1.Win10環境：那個檔案linux剛開始跑不出exe，後來使用windows cmd才順利執行。
2. 原本誤會題目意思，以為邊界的部分也可以亂下，後來看到討論區的發問時，只剩幾天要重新跑結果。
3. 原本想用RL去寫, 已經查了資料還有寫出一點架構＝＝？，發現RL的方法會超出時間限制，所以就打消念頭
4. 最後寫好的結果，雖然用了蒙地卡羅方法，已經各種修改想要增加勝率，但是還是會有30%機率輸給最差的random

Ideas of future investigation

如果未來要繼續研究相關議題，我們想要嘗試類似AlphaZero的演算法，一樣以MCTS為基礎，但是每個盤面的價值並不是透過隨機的playout來求得，而是透過由policy net和value net組合而成的深度學習網路來求得每個版面的policy和value，透過改良的UCB公式來做計算，並且透過自我對奕來更新網路裡面的weight，我們認為這樣應該可以輕鬆擊敗傳統的alpha-beta pruning或是MCTS等等。

但是礙於硬體的限制，沒有辦法用原本AlphaZero論文中提到的網路來做training，必須要自己改良比較不需要消耗那麼大的硬體的網路。