Introduction to Artificial Intelligence HW 3 Report

Team 21

0616309 王祥任, 0616070 陳柏諺, 0616206 吳冠潔

Abstract

本次作業的目的為以團隊的形式，嘗試實作可遊玩具特殊規則的黑白棋的 AI，可採用各類型的演算法，諸如 Minimax Tree, MCTS, 或 Learning 等。而在本次作業的實作中，我們最終採用 UCT 演算法搭配些許的 Domain Knowledge 來完成。

Difference with Normal Othello

本此實作中所遊玩之黑白棋，相較於正常的黑白棋有以下特殊之處：

1. 棋局開始時，盤面為清空狀態。
2. 四角不可落子。
3. 盤面中心 6x6 為絕對可落子的空間。
4. 雙方玩家皆只有 5 秒的思考時間。

The Algorithm We Choose

- ***UCT (Upper Confidence Bound Apply to Tree)***

即蒙地卡羅樹搜尋(MCTS)和信賴上界(UCB)的結合。在課堂上教過 Minimax Tree Search、Alpha-Beta Pruning 和 MCTS 等作法，皆可以使用在本次的作業中，但考量到：

1. 雙方玩家皆只有 5 秒的思考時間。(這是最重要的原因)
2. 8x8 的盤面大小並不能說是小規模的空間。
3. 組長比較想做 MCTS 相關(他覺得 Alpha Go 很厲害)。

所以最終，我們選擇了可以在任何時候停下來(Any-Time)且剪枝能力不錯的 MCTS 為主題來研究。而在搜尋有沒有 MCTS 和黑白棋的相關實作時，我們發現有人[1]以 UCT 演算法實作，故我們決定以此為範本實作本次作業，並加入我們針對本次作業的 Domain Knowledge 和特定的改良。

How It Works - Introduction to MCTS

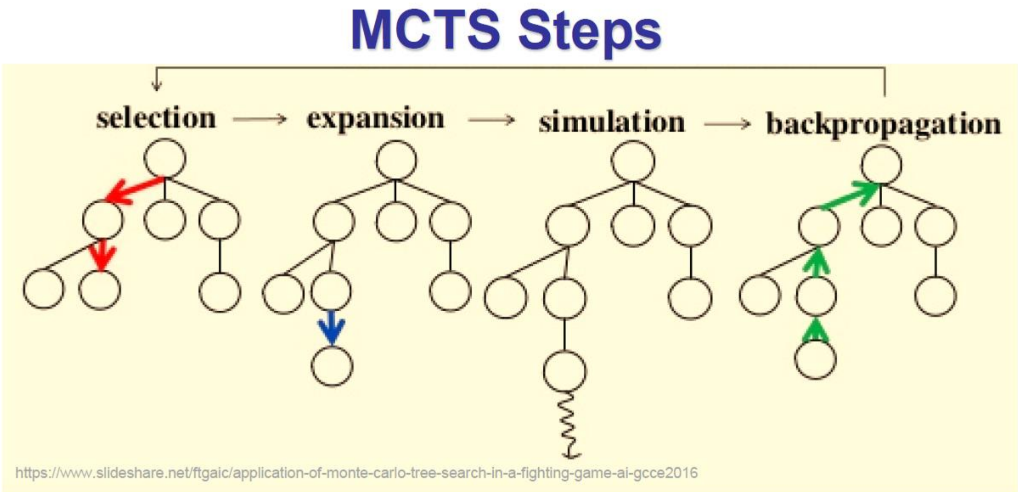

Fig1: Steps of MCTS from Slide #4

- ***MCTS Steps and How It Works***

由於課堂上已有講解，所以不贅述過多。UCT 演算法算是 MCTS 的一種特例，所以我們也必須先了解 MCTS 在做什麼。首先依照某一種條件，從根結點開始往下選擇(Selection)，若遇到未完全展開的節點，則將其展開(Expansion)，並用亂數隨機的方式將該展開後生成的節點模擬至終局狀態(Simulation)，並將最終利益沿路回傳、更新，直至根節點為止(Backpropagation)。

而 MCTS 運作的概念，就是依靠選擇優秀的節點和大量的模擬取得每一步的利益，並不斷更新、累積既有的資訊，從而大量剪枝，增進搜索的效率。所以如何去調節 Selection 就變得很重要了。

而 UCT 的特殊之處即在於 Selection 的階段時，以 UCB 公式做為選擇的依據。

$$\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

Q:在本次實作上為該點模擬後的勝利場數。

N:在本次實作上為該點被經過的次數(即被模擬的場數)。

c:主要去調節的參數，理想值為 1。

這個公式可以分成兩大項來看，左項(Q(v')/N(v'))，可以解釋為單純的勝率。而右項的目的，是為了顧及被冷落的節點(即勝率較低的節點)，當除數越來越大(即被訪問次數越來越多)，右項的值就會越來越小，就會給那些勝率較低的節點有機會被選到。

How It Works - Our Implementation of UCT

Algorithm 2 The UCT algorithm.

```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))

function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v

function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'

function BESTCHILD(v, c)
    return argmax_{v' ∈ children of v} Q(v')/N(v') + c·sqrt(2 ln N(v) / N(v'))

function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s

function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

Fig2: Pseudo Code of UCT (Without Our Modification) from [2]

- ***UCTSearch (Has Some Modification)***

進行搜尋的進入點，會回傳當下最佳(勝率最高)的行動。可以看到它會在剩餘時間足夠的情況下不停的進行 Select、Expansion、Simulation、Backpropagation 等 MCTS 行動，以求盡可能的趨近於真實的最佳解。

其中在主要搜索開始前，Create root node 部分，我們為了增加搜索樹的重複利用率，會試圖在舊有根節點的子節點中尋找是否有同樣的盤面。且在主要的搜索結束後，會將根節點更新為搜索結果中的最佳子節點(即包含回傳的行動的子節點)。但為了避免過度的比對和浪費，若無法在一層子節點中找到，則直接重新建立一顆樹。

因為本次作業並不會給予兩位玩家每一步的位置，所以只能直接用比對的方式。而這個方法最大的問題就是，若我方空過，則就無法在舊有根節點的一層子節點中找到相同的盤面。但在我們和猴子(全隨機落子)的試驗中，因為中央 6x6 為絕對可落子空間的緣故，所以玩家空過的機會就變得很小了，只有當盤面接近尾聲的時候，絕對可落子空間耗盡，才有機會出現空過，但此時盤面可能的變化已經縮減了不少，重新開始一顆搜索樹並不會損失太大。

- ***TreePolicy***

即 MCTS 中 Selection/Expansion 的部分，假設子節點已經全部可能都展開完畢了，則進行 Selection，並繼續向下找尋，直到找到尚未展開完畢的節點，則進行 Expansion，展開生成一個子節點，並將該子節點回傳。

- ***Expand***

即 Expansion 實際上執行的內容，隨機挑出一個可展開的子節點，展開、生成，並且回傳。

- ***BestChild (Has Some Modification)***

即 Selection 實際上執行的內容，也是 UCT 演算法在 MCTS 的基礎上最大的變動。在此處，我們計算某節點的所有子節點相對應的 UCB 值，並選擇具有最大值的節點繼續發展，由於 UCB 計算公式的設計，所以並不會只單純依照勝率來選擇，可以顧及其他被選次數不足的點，讓他們也有進行發展的機會，搞不好會是更好的節點。

而在此處我們加入了一些 Domain Knowledge，根據某位組員的說法，黑白棋重要的是落子的位置，而非單次翻棋的子數。以典型的黑白棋來說，最有利的位置為四角，當落子在此處則絕對不會被翻轉。而以本次作業的規則而言，我們對盤面施予了以下的權重。

	0	1	2	3	4	5	6	7
0		2.5	1.0	1.15	1.15	1.0	2.5	
1	2.5	0.85	0.9	1.1	1.1	0.9	0.85	2.5
2	1.0	0.9	1.12	1.07	1.07	1.12	0.9	1.0
3	1.15	1.1	1.07	1.03	1.03	1.07	1.1	1.15
4	1.15	1.1	1.07	1.03	1.03	1.07	1.1	1.15
5	1.0	0.9	1.12	1.07	1.07	1.12	0.9	1.0
6	2.5	0.85	0.9	1.1	1.1	0.9	0.85	2.5
7		2.5	1.0	1.15	1.15	1.0	2.5	

Fig 3: Weight of the Board (By Ourselves)

```
double weight4[] = {
    0.918386, 1.100000, 1.056350, 0.983453, 0.983453, 1.056350, 1.100000, 0.918386, // r0
    1.100000, 0.900000, 0.925536, 0.990398, 0.990398, 0.925536, 0.900000, 1.100000, // r1
    1.056350, 0.925536, 0.976745, 1.006742, 1.006742, 0.976745, 0.925536, 1.056350, // r2
    0.983453, 0.990398, 1.006742, 0.998468, 0.998468, 1.006742, 0.990398, 0.983453, // r3
    0.983453, 0.990398, 1.006742, 0.998468, 0.998468, 1.006742, 0.990398, 0.983453, // r4
    1.056350, 0.925536, 0.976745, 1.006742, 1.006742, 0.976745, 0.925536, 1.056350, // r5
    1.100000, 0.900000, 0.925536, 0.990398, 0.990398, 0.925536, 0.900000, 1.100000, // r6
    0.918386, 1.100000, 1.056350, 0.983453, 0.983453, 1.056350, 1.100000, 0.918386, // r7
};

double weight4_black[] = {
    1.057874, 1.100000, 1.058360, 0.981983, 0.981983, 1.058360, 1.100000, 1.057874, // r0
    1.100000, 0.900000, 0.924141, 0.981270, 0.981270, 0.924141, 0.900000, 1.100000, // r1
    1.058360, 0.924141, 0.979780, 1.005995, 1.005995, 0.979780, 0.924141, 1.058360, // r2
    0.981983, 0.981270, 1.005995, 1.000843, 1.000843, 1.005995, 0.981270, 0.981983, // r3
    0.981983, 0.981270, 1.005995, 1.000843, 1.000843, 1.005995, 0.981270, 0.981983, // r4
    1.058360, 0.924141, 0.979780, 1.005995, 1.005995, 0.979780, 0.924141, 1.058360, // r5
    1.100000, 0.900000, 0.924141, 0.981270, 0.981270, 0.924141, 0.900000, 1.100000, // r6
    1.057874, 1.100000, 1.058360, 0.981983, 0.981983, 1.058360, 1.100000, 1.057874, // r7
};
```

Fig 4: Weight of the Board (By Competition between AIs)

當該回合的行動是下在該點上時，UCB 的值便會乘上相對應的權重。以最有利的八個點開始往回推導，給予每個點可能的權重。而在我們的 AI 與猴子(全隨機落子)對戰的情況下，比起不給予權重，給予權重者的確有較高的勝率。

而 Fig3 的權重，是由我們的組員自行推測的。而 Fig4 的權重(也是現行使用的版本)，為 AI 內戰得出的結果，具體於稍後的討論補充。

- ***DefaultPolicy***

即 MCTS 中的 Simulation，以全隨機落子的方式，將遊戲快速的模擬到終局狀態，並回傳其結果，當同一節點模擬的次數越多，其結果理論上會越貼近真實的結果。

- ***Backup***

即 MCTS 中的 Backpropagation，從葉節點開始一路回至根節點，所有路上經過的節點，N 值加一，而根據模擬結果，假如最終為白子勝利，則所有白子節點 Q 值加一，而黑子節點 Q 值不須變動。

而 Backup 的 Pseudo Code 中，我們有找到一種特別給零和遊戲使用的更新方式，若白子勝利，則白子節點 Q 值加一，但是黑子節點 Q 值減一。兩種做法我們都有進行嘗試，在思考時間較短的時侯，黑子有變動的更新方式偶爾勝率會較高，但當思考時間拉長後，兩者勝率的差異並不明顯。所以最終我們決定採用原先的更新方式即可。

Some Discussion

- ***Score and Parameter***

本次作業對於排名的計算方式上，是以差距目數作為單次戰局的分數(Score)，假設我想在此種機制下取得最優的分數，則不應假設為零和遊戲，而是以分數做為利益的參數來運行。

但實作中，我們定義此黑白棋為零和遊戲(Zero-Sum Game)。會如此設定是因為 UCB 的參數(c)設定。我們並沒有足夠的時間和足夠強大的對手(我們只有猴子(全隨機落子)和一個比較會下黑白棋的組員)可以讓我們去調整成最佳的參數(c)，所以我們決定直接採用 UCB 的理想值$\sqrt{2}$ (c=1)。

- ***Smartest Opponent or Stupidest Opponent***

假設今天我手持白子，則在 Selection 階段時，我在選擇黑子時，應該選擇 UCB 最小者(某種意義上的最低勝率)還是依舊選擇 UCB 最大者?

由於我們並不能操控對手下棋，所以我們應該考慮的是最糟的情況，即對手也會盡可能的下在最

好的位置上，故我們應該選擇 UCB 最大者。在搜尋的時候，過程應盡可能的貼近盤面的現實狀況，而非一味的追求理想解。

- ***Training of Weights***

這是當我們完成初版 AI 後產生的想法，初版使用的權重是我們的組員手動去推導出來的，然而他自己也不確定是否正確。於是我們產生了一個想法，讓兩個 AI 互相對打，根據戰果和他們所下的位置，更新出一份自己的權重。

更新的方式相對粗糙，假設今天黑子勝利，則這場黑子所有落子的位置，權重皆增加一常數大小，而白子方面則減少一常數大小。並使用此更新過後的值，再次對戰、更新。

有趣的是，根據給予思考時間的不同，某些點的重要程度會有所不同。但不變的是，(1, 1)、(1, 2)、(2, 1)、(2, 2)和其輻射對稱所對應的點，重要的程度永遠會是最低的，而(0, 1)、(1, 0)和其輻射對稱所對應的點，權重永遠會是最高的。這和我們當初推測的大致相同。

還有一個比較特別的觀察是，在 AI 內戰的情況下，執白子(即後手)的勝率會大於執黑子(即先手)。

另外，權重的區間也是值得討論的部分，而根據和猴子(全隨機落子)的結果，將權重的區間調整為[1.1, 0.9]是目前所能得出最佳的結果。這一部分會根據 UCB(未和權重相乘前)的具體數值而有所變動，假設權重的區間過大，則 UCB 值很可能無法發揮其作用，轉而完全交由權重主導，這就本末倒置了。

Conclusion

- ***The reason why we use UCT and some Thoughts***

最一開始是選擇以 MCTS 為主題實作，原因不外乎是思考時間的限制還有個人興趣。

而在我們翻到的資料[2]中，我們得知 MCTS 本身其實算是比較弱的演算法(對較複雜的遊戲而言)，所以必須加入一些強化的機制，例如 Domain Knowledge 或是重複利用的機制等，所以我們在 Selection 方面加入地圖的權重，而對猴子(全隨機落子)的勝率的確也有所上升，而由於重複利用的機制需要一定的思考時間才能發揮功用，所以沒辦法和猴子下太多次，無法驗證，故僅做為無傷大雅的嘗試加入。

而資料[2]提到說 UCT 做為 MCTS 最常見的變體，且也有不錯的表現。另資料[1][3]是我們找到的資料中有最貼近的實作者，同樣都是採用 UCT。故最終我們的目標就鎖定在時做 UCT 演算法上。

由於我們都是第一次接觸這類 AI 的撰寫，而且測試的對象只有猴子(全隨機落子)和其中一位比較會下黑白棋的組員，所以我不能說它的表現能夠勝過其他組別，但做為一次嘗試，能夠寫出一個比猴子還厲害的 AI，並更加了解 MCTS 和 UCT 已經是收獲良多了。

Reference

[1] C++程设实验项目三：黑白棋与基于 UCT 算法的 AI

[2] Monte Carlo Tree Search

[3] UCT（信心上限树算法）解四子棋问题——蒙特卡罗法模拟人机博弈

[4] IntroAI_Set04.pdf