2021 電腦對局理論 期末報告

電機碩一 **R10921071** 方文忠

目錄

一.實作說明

Ver1　Bitboard + Nega-scout 版本

下表為 depth 同樣等於 3，兩個演算法所花時間的實驗記錄

實驗次數	baseline (Nega-max)	Nega-scout + bitboard	diff
第一手	15	11	+4
第二手	16	7	+9
第三手	12	7	+5
第四手	9	5	+4
第五手	10	4	+6
第六手	8	4	+4
第七手	8	4	+4
第八手	7	4	+3
第九手	6	2	+4
第十手	4	2	+2
Total	95	50	+45
Avg	9.5	5	+4.5

單位:秒

加入 bitboard 與 Nega-scout 後平均每一手加速 4.5 秒

而 5 戰的結果則是不出意外的 5 和，因為搜尋層數一樣，僅僅是 Nega-scout 能比較快的計算完成而已，接下來希望能藉由增加層數來將時間轉換成勝率

小結:

但在開局中光是加深為 5 層，時間就會爆發性的成長，因為開局時 chance node 太多了，並且每多一個 chance node 在開局大約就要試算所有棋子的可能性，因次每層約會多 32*14(兵種)個節點，如果這樣的層數有五層就會非常巨大，然而在開局計算大量的 chance node 其實沒有意義，當翻開時機率都會收斂成一個，因此過多計算是沒有太多幫助的。有鑑於此，應該先制定 open strategy 等開局策略，以渡過這段 Chance node 計算爆發期。

Ver 2 開局策略(open strategy)版本

開局策略一開始想使用 Rule Base，比如對方翻到王，我方就翻炮位或是王周圍賭兵卒，但是除了第一手以外情況可能很複雜，比如王旁邊一定要賭翻兵卒，可是若翻開暗子的旁邊就是敵方的其他車、馬等，那很可能即使賭對了也會被吃掉，因此我決定在開局階段，只搜尋明子周圍和炮攻擊位的棋子，讓搜尋法自行給出

答案。

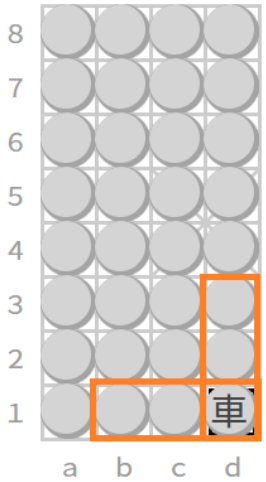
開局規則

如果是第一手先手，採取保守策略，翻角落的子

如果是第一手後手，則視對方子作不同反應。

只考慮鄰子

在開局階段，我覺得只有在各明子附近的暗子才有思考價值，比如一開始在角落有明子，則對方只有翻到該明子旁邊或包位，才有可能對局面有影響，如下圖，只有翻到橘框範圍內的子可能吃掉黑車/被黑車吃掉，進而影響局面分數，反之其他地方比較像另開戰場。



而找明子周圍相鄰的，就視為不分敵我對明子做的 bitboard 走步 Expand，只是這次要取走步的終點當作翻子的起點，而炮位也只需要將 bitboard 炮位吃子 Expand 對明子做即可。

實驗結果

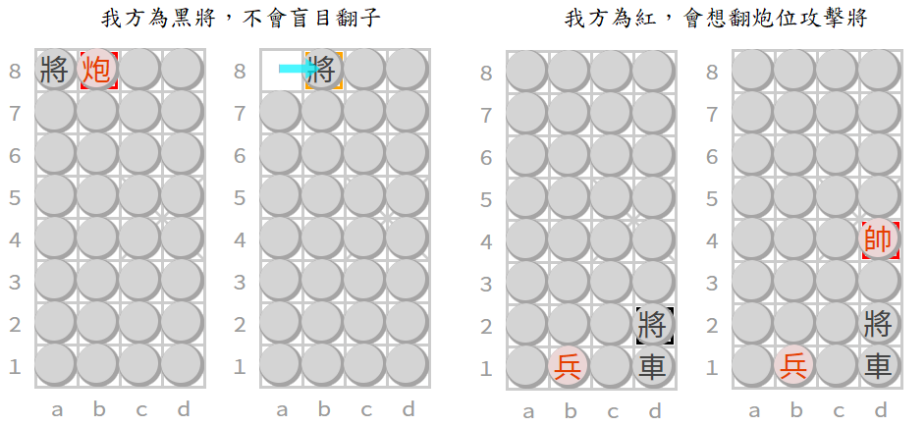
以下為實驗三場深度為 3，加入開局策略的運算時間，每一場紀錄第十手、終局時的剩餘時間

場次	baseline (Nega-max)	Nega-scout + bitboard+ open strategy
第一場 第 10 手	851	895
第一場 終局	801	878
第二場 第 10 手	840	898
第二場 終局	799	891
第三場 第 10 手	863	896
第三場 終局	839	882

單位:秒

可以發現比起 Version1 大概十手時間提升了 10 倍左右

而效果方面，嘗試了 8 場也是 8 場和局，並且有觀察到下面的現象



因此此法現階段應該可行。

當暗子少於 10 個時，會再進入窮舉考慮，因為此時剩下可能性不多，不會消耗太多時間。

小結:

將層數增加為 4 層時雖然算的完了，但仍需要 30 秒左右的時間，因為搜尋時間會隨深度呈指數成長，很可能 5 層需要 900 秒的時間，很顯然是不能被接受的方案，因此下個版本想加入同型表、chance node search 等方法再進行加速。

Ver 3 同型表(Transposition Table)版本

寫同型表時出了不少 bug，包含 hash table 開不夠大、chessboard 的 hash key 在 copy constructor 中忘記寫入...等等，是目前花費時間意外最久的，但效果而言加速的比想像中多，以下是三場深度為 3 的實驗記錄，由於速度太快，只記錄終局所花時間，並與上一版本比較

場次	Nega-scout + bitboard+ open strategy	Nega-scout + bitboard+ open strategy + transition table
第一場 終局	878	898
第二場 終局	891	898
第三場 終局	882	898

單位:秒

可以觀察到總時間提升了 8 倍以上，並且可以在合理的時間內跑完 7 層左右了

合理的選取 Transposition table 大小

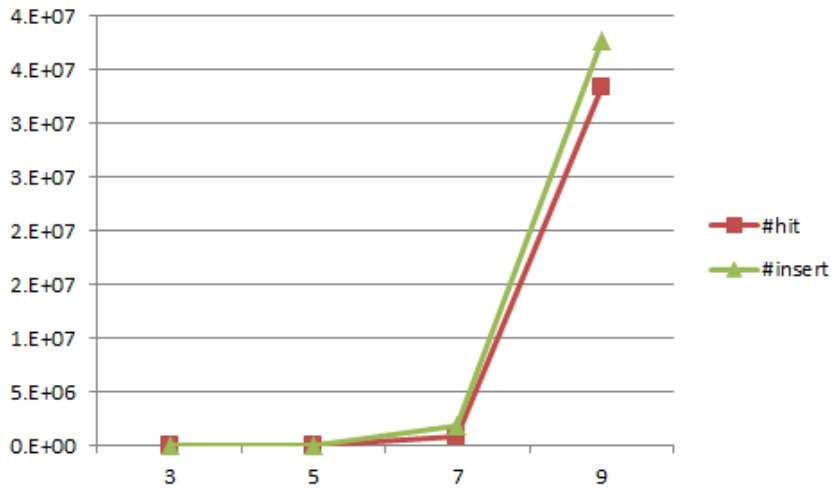
同時，我有紀錄 insert 與 hit 的次數，並以此決定 Transition table 目前大概需要開多少

以下實驗為使用上一次作業的相同測試盤面(全棋子翻開，包含炮)

Depth	#hit	#insert
3	65	3028
5	27689	88145

7	915520	1901115
9	33388172	37654376

單位:次數



可以觀察到 insert 次數約略比 hit 次數多，但兩者每加身兩層大概會增加 20 倍以上，而我們在意的應該是 hit 次數(才能幫我們節省搜尋時間)，因此我假設 final project 先以 9 層為目標，並且加上 chance node 會有更多消耗，因此選用 2^28 次方作為 transposition table 的大小，也就是大約 2.6E+8，盡量合理的使用記憶體資源。

小結:

接下來就是實作 Star 系列針對 chance node 的搜尋演算法，期望能速度能有更進一步的提升。

但是加入 transposition table 可能會導致循環盤面發生，因為 transposition table 沒有加入歷史資訊，即使先行比對，也會算不到對手的循環走步。

Ver 4 翻子策略 與 star 演算法版本

翻子與動明子我覺得就像 explore 與 exploitation，翻子存在一定程度的風險，而且如前面所說，翻子計算成本巨大(考慮不同可能性)，但成效可能不高(只有其中一種可能會發生)，而 Star 與翻子策略就是希望解決這兩個問題。

star 演算法

類似 MCTS 的 progress pruming 的概念，當一個翻子位置極度沒機會時，就不用搜尋它，以此降低計算的節點數，而也類似於 progress pruming 的概念，因為我們設計的審局函數考慮太多動態的東西，很難事先透過 normalization 壓到[-1,+1]，因此 Vmin , Vmax 也是我們自己推估的，而[Vmin,Vmax]區間若太大，則會退化成暴力求取期待值的 star0 演算法，反之若[Vmin , Vmax]區間太小，則可能使正確答案被切掉，因此好的 Vmin , Vmax 可能需要透過實驗去得到。

以下是推估 Vmin = -5000 , Vmax = 5000 的實驗結果
深度為 7 層，隨機觀測五手發生 star cut 的次數

Star Cut 發生次數	總共搜尋 Nodes
193	7607
59	24953
209	30240
2479	73751
40	7317

我覺得發生次數有點太少，可是不太敢進一步縮小範圍擔心 cut 掉有機會的翻子，因此最後將範圍設最大以趨近於 star0。即是題目程式原本舊有的版本加上考慮 beta cut 的 Nega scout 版本。

翻子策略

翻子與走子就像 explore 與 exploitation，而我比較偏向穩定的 exploitation，因此我將只在根結點考慮翻子的走步，並且翻子的走步會考慮較為淺層，以加快思考速度與將重心放在明子上。

小結:

經過上述改良，已經可以計算至 10 層左右，因此可以嘗試將計算時間轉換成計算深度，以利提升勝率。

Ver 5 逐層加深與 Iterative Deepening Aspiration Search (IDAS)

逐層加深是為了防止搜尋樹不平衡時，答案在淺層，我們卻從深層處開始搜尋導致浪費計算成本，而 IDAS 則是類似 scout testing 的概念，以上一層為依據，建立在假設"審局函數變化隨層數變化不會太大"，去縮小 scout alpha , beta 的範圍，並且如果測試失敗了，就重新搜尋一個精確的解。

也和 star 演算法的 Vmin, Vmax 一樣，當分數不是正規化的時候需要去調整一個好的 threshold，若 threshold 太大，則退化成普通的 Nega-scout，若 threshold 太小，則會每次都 failed 並搜兩次，因此選取好的值很重要。

小結:

將深度加深後，與 baseline 比五場為 1W4D0L，並沒有很顯著的進步，觀察棋局都會偏向大優而無法贏下，因此需要將作業一的 Evaluation function 搬過來並進行改良。

Ver6 對暗子之審局函數 Evaluation with dark piece

審局函數也是經過多次的調整，以下會說明每次調整的內容

1.移植前幾次作業的內容

前幾次作業其實已經處理到無暗子的各種盤面，並且已有不錯的表現，因此可以移植過來使用

* 追擊系統

因為每一步分數都一樣(大優)，所以很難得到正確的走步，因此加入曼哈頓距離去引導我方子力靠近對方子力，並且在 distance == 2 且 col == 1 && row ==1 的角位分數應該要最高，否則會變成一直長抓子

* 天敵分數

依場上所剩天敵去動態調整自己的分數，這樣會讓程式趕於換子製造簡化優勢局面(如將士對兵士 簡化為 士對兵)

* 大局觀

會先依雙方子力差距對分數進行一些調整，例如:對方有絕對比我方多的子，且我方無包，則我方分數應比和局分數還低，才會追求和局，也會敢於在"將"還在的時候棄大子去換兵以製造必勝局面

* 包卒動態調整

包和卒可謂是暗棋最特殊的兩個兵種，包的吃子與走步不同之外，能吃所有其他子，卒則是輸給其他子，但可以擒王，因此需特別為它們制定分數，比如與卒對方王的距離、對方有王時卒需要保留...等等

移植版結果: 12W6D2L

2. 避免和局

目前情況除了運氣不好之外，仍發生不少和局，究其原因發現都是大優不會贏，因為太多走步都會贏，導致所有走步分數都一樣，又因為走步排序，導致我們只會重複同樣的走步，最後和棋，因此加入以下修改

* 考慮深度

將勝利的分數- depth，這樣會使各深度的勝利走步出現差別，並使 AI 選擇深度較淺(較快)勝利的走步

*和棋分數調整

將不吃翻和棋分數訂為靠近 0(勝局、負局中點)，而重複盤面和局分數則訂為靠近負局，這樣平時決不會輕易走，但快輸時仍會接受和局

避免和局結果: 15W5D0L

3. 微調

*上一手吃子

因為審局函數盤面沒有時序資訊，因此加入上一手的吃子資訊，會鼓勵吃子，同時鼓勵保護自己的子

*mobility

在審局時本來就會有 legal move 的判斷，就是用這個加上一點分數，再依是否是我方決定正負號，這樣會鼓勵將對方子圍困

*平滑針對暗子審局函數

用 0 的方式加上 static score，這樣翻子時可能會出現分數極端的震盪，會使程式偏向翻子的行為，因此我將分數判斷改成 (總分 - 被吃掉的子的分數)，這樣會翻開的子被吃或是吃子時分數才會改變，也就才有翻子的意義。

微調結果: 19W0D1L

Ver7 Time Control

採用每步常數時間，因此會先透過實驗取得每局平均的步數，以下是十局棋的時間實驗記錄

局數	每局總步數
1	117
2	93
3	96
4	287
5	81
6	113
7	165
8	79
9	142
10	91
AVG	126.4

(單位: 步數)

因為共有 900 秒的自由時間，平均有 126.4 步，因此大概每手取 6.5 秒，並且限制最大深度在開局階段為 10 層，中局階段為 12 層，殘局階段為 14 層

而階段是以場上剩下子數判斷，也是一個可調超參數。

Ver8 Move ordering & refutation table

Move ordering

Move ordering 會大大影響 alpha-beta cut 的效率，所以十分重要，本次改良作業一的走步排序，考慮行動 吃子 > 走子 > 翻子，並且都由大子開始行動，同時當對方王存活時，考慮 王、士、兵卒、包、象、車、馬的吃子走步，而對方王被吃掉後，則兵卒順序移至最後(王、士、包、象、車、馬、兵卒)，實裝後肉眼

觀察每層搜尋節點數有略為下降，同時翻子也會優先考慮翻出強子，讓 star1 更有機會發生 cut。

Refutation table

結合 Iterative deepening，記錄 PV path,並作為下一層改善 move ordering 的依據，但可能是暗棋依照上述 move ordering 就有不錯的效果，加入 refutation table 感覺並無太大的差別。

Ver 9 Dynamic Search Extension

用於避免不穩定的盤面，原本嘗試當有對捉子時就自動延伸一層，但發現運算量會太高，最後改成王被捉時才會增加層數，但是或許因為暗棋的子移動範圍並不大，不容易出現太激烈的換子局面，因此可能幫助不大。

版本記錄總結

目標	結果
version 1 bitboard + nega-scout	
加速: 減少 Expand 計算成本 減少 Search 節點	每一手約提升 4.5 秒
version 2 open strategy	
加速: 透過規則跳過暗子最多的前兩手 減少需思考的暗子數	(較前版本) 以三層而言，每一手約提升 10 倍速度
version 3 Transpoition table	
加速: 減少重複搜尋時間	(較前版本)以三層而言，總時間約提升 8 倍速度()
version 4 Star 1 與翻子策略	
加速: 減少暗子分支 Search 結點數 減少暗子搜尋深度	皆會發生 cut，但由於隨機性，發生 cut 次數不定，開中局可以計算到 10 層了
version 5 Iterative deepening & IDAS	
加速+提升勝率: 加深搜尋深度，並預防"噴水池效應" 類似 Scout 的加速深度計算方法	能在 5 秒左右搜至 8~10 層 但勝率還遠待改進
version 6 Evaluation with dark piece	
提升勝率: 對盤面估計更為精確	對 baseline 勝負達到 19W0D1L
version 7 Time Control	
控制時間資源	
version 8 Move ordering & refutation table	每層需搜尋節點數下降
加速: 提升剪枝的機會	
version 9 Dynamic Search Extension	
提升勝率 防止搜尋停在激烈的位置，造成嚴重誤算	無觀察到明顯差異

二.實驗

(一). 不同設定時間對勝率的影響

以下測試常數時間對勝率的影響(vs baseline)

每步秒數	結果	勝率
3.5	17W3D0L	85%
4.5	18W1D1L	90%
6.5	17W3D0L	85%
8.5	18W2D0L	90%

可以觀察到或許每步秒數越高可能會更穩定，但也有可能是因為與 baseline 比較所以秒數低走出漏著也不會被懲罰。

(二).比賽結果

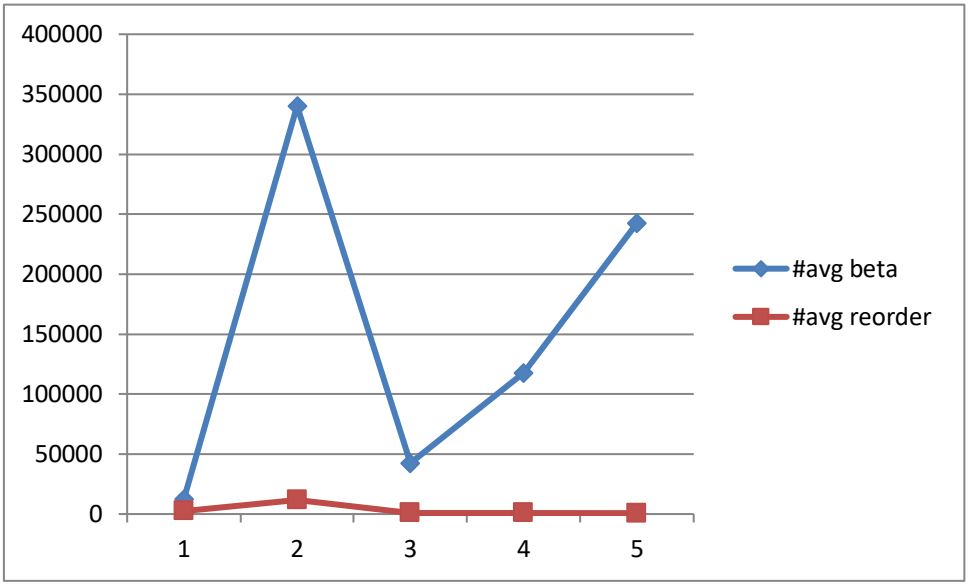
	第一場	第二場
第一局	W	W
第二局	L	L
第三局	L	L (大優送將，加上最後 illegal)
第一天總結: 發現兩個 bug 容易送將與 illegal move 同時遇到對手很強 QQ bug fixed: 和棋判斷沒加入 move 早上新改的 Iterative deepening 沒有寫到正確的走步....		
第四局	W	W
第五局	W	W
第六局	W	W
第二天總結: 很幸運昨日 bug 解決之後本日全勝~ 但很可能是因為昨天輸太多，今天對手比較弱 週一就將挑戰當前第一名		
第七局	W	D
第八局	W	L
第九局	W	L
最終名次	11W 1D 6L 第四名	

三.討論

(一)refutation table heuristic 之效能討論

以下為裝上 refutation 後的各局 50 手平均發生的 beta cut，與使用到 refutation 的 reorder 的次數

#avg beta	#avg reorder
12294.66	2544.31
340168.83	11714.44
42436.76	1076.78
117563.92	909.3
242393.18	724.4



可以觀察到 beta cut 的發生與 reorder 有正相關

次數	#avg beta with refutation	#avg beta without refutation
1	12294.66	29767.3
2	340168.83	44819.66
3	42436.76	74459.84
4	117563.92	42853
5	242393.18	202729.68
平均	150791.5	78925

比較沒有實作 refutation table 與有實作的差別，可以發現 beta cut 平均的出現次數增加了 2 倍之多，並且 refutation table 可能更早發生 beta cut，總體來說可以提升速度。

(二). alpha-beta 與 MCTS 之結合討論

以下將從兩個方法的兩個特點進行討論，MCTS 的不用人為制定審局函數特性，與 alpha-beta (scout 版本)中使用 scout 剪枝加速

1. MCTS 完成殘局庫

MCTS 的優勢之一在於不需人為制定審局函數，可以藉由大數據抽樣去捕捉到理想的審局函數，但由於棋局的複雜度太高，需要花長時間與成本去模擬，所以會有誤差或是在一定深度仍需要人為制定審局函數給予引導，同時這點也可能是劣勢，MCTS 相比 alpha-beta search 更缺乏解釋性，這也代表了更難去微調參數，我覺得如果在像這次比賽需要短時間動態調整參數、模型時會成為一個問題。

綜上所述，我覺得可以以 Alpha beta 為主幹，用 MCTS 訓練殘局庫，因為殘局的深度可能比較低，並且這也是可以 Offline 完成，所以不用太在乎即時性，並且殘局可能比起中局有更多審局規則要寫，使用 MCTS 或許可以省下這部分的心力。

2. Scout 應用於 MCTS 剪枝

Alpha-beta 的優勢之一則在於最多能達到 O(根號 n)的搜尋時間，n 為所有對局樹的節點數，對比賽時間有限時很有用，其中一個技巧就是 scout，即先檢測是否值是我們要的範圍內，如果沒有就可以剪枝，如果有則再 re-search，這就像快取中的 hit 機制一樣，是一個期待值上有利的賭博，而 MCTS 則只有 Expansion Policy,RAVE 等以信心決定該節點是否展開的機制。

綜上所述，我覺得可以在 MCTS 中達到一定深度就交給一個簡單的審局函數判斷這個 path 有沒有值得繼續模擬的必要，如果沒有可能可以回傳失敗的審局函數值(就像 soft failed 版本 alpha-beta 一樣)作為模擬分數，這樣雖然失去了一點 MCTS 不用人為訂審局函數的優點，但應可以加快搜尋速度，而比賽往往會限時，再加上 MCTS 本質就需要以量取勝，因此加入這種剪枝可能可以提升 MCTS 效果。

這兩個方法本身也具有類似的地方，如 star1 的信賴區間 cut 就類似於 progressive pruning、翻子與走子的審局函述取捨類似於 UCT 公式的探索與開發...，兩個方法也有很多技巧是都可以使用的，如同型表、pondering 等...。

(三). 電腦對局程式的其他目標

電腦對局程式除了獲勝以外，還有很多目標尚未完成，以下試著提出四個目標

1. 證明先手結果

如老師上課所說，即使現在 alpha 系列能夠宰制人類棋手，但圍棋是否先手必勝等本質的問題仍未解決，電腦對局程式既然能夠有高深的棋力，應可以輔助人類找到此問題的答案，未來可能可以想辦法結合數學證明的方法，甚至可能可以"有效率的"窮舉得到證明。

2. 輔助人類練棋

本身平時有在藉由棋軟練棋，現階段的棋軟僅僅只能告訴我們哪一步是妙手，頂多輔以分數讓我們復盤時發現自己哪一手走漏、走軟，讓我們類似 Reinforcement learning 的方式依照分數去改進自己的下法，但未來或許可以讓電腦程式提供更具解釋性的理由，讓人類能夠歸納原則以及更好的記住盤面，比如:"一車十子寒"、"寧失一子不失一先"，這種類似順口溜就是前人總結出關於象棋的智慧，甚至可能可以擔任教學者的角色，讓新手能有一條平滑的訓練曲線，抑或是依照棋手個人棋風進行開局推薦....等等，如何將電腦運算出的知識傳承給人類，我覺得是很好也很重要的目標。

3.研發新棋種

未來可能可以從各程式思考棋類的方式，讓人類試著發現"為甚麼那種棋有趣"、"那種棋的精隨"...等，比如暗棋我們著重設計的地方就在翻子、包卒的審局函數處理，某個角度代表了暗棋翻子具有賭博性、包的走法吃法不同帶來、卒可以反吃帥等等特殊規則帶來的趣味性，或許可以藉此得到新遊戲的創意。

4. 模組化並應用於其他問題

每解決一種棋種，可能就可以應用於其他問題，像這次暗棋的翻子可能就可以用於麻將、撲克牌這種也有機率成分的遊戲，也像老師上課提過的，很多現實問題也可以看作是雙人棋類，比如 OS 的排程問題，而如果能把對局程式的分析成小模組，並且找到能適用的場合，或許就能有更廣泛的應用性。

四.結論與心得

理論方面，本門課非常精彩，光是主搜尋方法從一開始最簡單的 Min-Max search tree 不斷往下延伸，了解到了 alpha-beta serach，Nega-scout search、chance node searhc，等等進階變體，以及其他 iterative deepening、bitboard、transiposition table 等等有趣且有用的技巧，此外也花了 1/3 學期在講另一種靠模擬的蒙地卡羅樹搜尋。這些方法絕對不只能用在對局理論上，也一定有機會可以運用在其他地方，我覺得更重要的是，從學習演算法與資料結構的概念中去了解怎麼最有效的解決問題，比如 bitboard、最終結合 alpha-beta search , scout , chance node search 的主程式，就是盡力節省計算成本下的作品，令我印象十分深刻。

實作方面，本門課非常的繁重，但也很有成就感，我覺得比起以往年度(藉由看往年網站得到的資訊)，本學期的這種漸進式的作業非常棒，每次都可以利用之前寫好的部分東西，最後期末 project 更是集大成，看見最終打敗 baseline、在比賽中取得不錯的成績時真的會很開心~，並且真的去寫過作業，我覺得比起考試幫助更大，光是做筆記、在腦袋中模擬，不如真的用手刻一個出來來的印象深刻。

課程安排方面，從同學分享作業一中學到很多東西，最重要的就是版本測試紀錄，還有很多審局函數的設計，也幫助我在後續作業中取得不錯的成果，比賽也如老師所說，是一個很好的實際測試，在第一天中我就發生了一個對 Baseline 不會看出來的 bug，導致四敗。此外也從中學到 gprof、平行化等等設計技巧，比較可惜的是前面老師說的很精采的 DFS 等部分沒機會聽到。

最後感謝老師與助教這學期的指導，也感謝很強大的同學與跟我熬夜趕工的戰友們，這是我本學期修到最有趣又紮實的課，在此落下帷幕。