國立臺灣師範大學
資訊工程研究所碩士論文

指導教授： 林順喜 博士

暗棋中棋種間食物鏈關係之探討與實作

研究生： 謝政孝 撰

中華民國九十九年六月

摘要

電腦棋類一直是人工智慧發展的重要領域之一，而電腦暗棋至今仍較少人對其做較深入的研究。暗棋是屬於不完全資訊含機率性的棋類遊戲，不像西洋棋、象棋是屬於完全資訊的棋類遊戲，所以如果用一般遊戲樹進行搜尋，在走棋與翻棋夾雜的情況下，會因分枝度過大而無法做深入的搜尋，因此難以做出較佳的決策。

本論文希望改良先前謝曜安研究生的暗棋程式，首先改進他的走步生成方式，與審局函數的計算。由於他的審局函數是採用靜態子力去計算分數，不論盤面資訊如何，其各個子力價值恆為固定，在許多情況下會產生誤判，我們希望可以藉由盤面改變而動態的改變子力價值，更客觀小心的審視盤面，並以這審局函數來實作在暗棋中關於其棋種間特殊的食物鏈關係，以期加強暗棋程式的棋力程度，並使棋力超越人類玩家水平。

ABSTRACT

Computer chess is always an important research area in artificial intelligence. At present, there is less paper dealing with the playing strategies of Dark chess. Dark chess is an incomplete information game with probabilities, which is not the same as complete information games, such as chess or Chinese chess. If we use conventional game-tree searching techniques to tackle Dark chess, then the number of branches will be very large because there are lots of moves for both “dark pieces” and “bright pieces”. Hence, it is not easy to improve the strength of the Dark chess program by using the conventional game-tree searching techniques.

This thesis is written to improve the Dark Chess program which was developed by the postgraduate Hsieh,Yao-An. We improve his move generator first, and then the evaluation function. As his evaluation function used static scores to calculate the materials’ values, regardless of how the chess game plays out, in many cases it will lead to wrong judgments. We want to dynamically change the chess materials scores when the chess board is changed to a more objective measurement. We carefully consider the unique food chain relations of the chess species and design a Dark Chess program to enhance the evaluation function. Finally, we combine several techniques to improve the strength of the program.

致謝

感謝指導教授林順喜博士，指導本論文之研究內容與寫作方向，並且教導許多演算法與人工智慧領域的相關知識。

另外還要感謝實驗室的其他成員——潘典台學長、魏仲良學長、黃士傑學長、黃立德學長、莊臺寶學長、趙義雄學長、謝曜安學長、劉雲青學長、白聖群學長、葉俊廷學長、黃信翰學長，同學詹傑淳、賴昱臣、陳俊佑、李明臻、李啟峰，以及學弟妹勞永祥、陳志宏、蔡宗賢、唐心皓。

特別感謝指導教授林博士，及白聖群學長、劉雲青學長，在平時提供實作技術上的建議與方向。

最後感謝我的父母栽培我，並適時地給予鼓勵與支持，使我生活無虞，讓我能順利完成學業，僅將此論文獻給我最敬愛的父母。

目 錄

摘要 ..... **i**
ABSTRACT ..... **ii**
致謝 ..... **iii**
目 錄 ..... **iv**
圖目錄 ..... **v**
表目錄 ..... **vii**
第一章 緒論 **1**
第一節 暗棋規則及玩法介紹 1
第二節 論文概要 4
第二章 資料結構 **5**
第一節 棋盤-棋子映射結構 5
第二節 著法預處理 11
第三節 著法產生流程 13
第三章 搜尋演算法 **14**
第一節 簡介 14
第二節 Min-Max 搜尋演算法 15
第三節 Nega-Max 搜尋演算法 18
第四節 Alpha-Beta 搜尋演算法 19
第五節 Transposition Table 23
第六節 允許空步 31
第七節 Iterative Deepening 32
第八節 利用食物鏈的關係與威脅度設計審局函數 33
第九節 翻棋策略 39
第四章 結論與未來方向 **41**
第一節 結論 41
第二節 未來研究方向 42
參考著作 **43**

圖目錄

圖 1-1 暗棋的棋盤.....1
圖 1-2 暗棋的棋子.....2
圖 1-3 長捉示意圖.....3
圖 1-4 長捉示意圖.....3
圖 2-1 一維陣列表示法.....5
圖 2-2 board 與 position 示意圖.....7
圖 2-3 current 示意圖.....8
圖 2-4 current 更新示意圖.....9
圖 2-5 MoveTab 示意圖.....11
圖 2-6 OrderTab 示意圖.....12
圖 3-1 井字遊戲對局樹範例.....14
圖 3-2 Min-Max 示意圖.....16
圖 3-3 棋類中 Min-Max 搜尋示意圖.....16
圖 3-4 Min-Max 搜尋演算法虛擬碼.....17
圖 3-5 Nega-Max 搜尋演算法虛擬碼.....18
圖 3-6 Alpha-Beta 示意圖.....19
圖 3-7 Alpha-Beta 示意圖.....20
圖 3-8 Alpha-Beta 搜尋演算法的虛擬碼.....22
圖 3-9 Nega-Max 形式的 Alpha-Beta 搜尋演算法虛擬碼.....23
圖 3-10 相同盤面示意圖.....24
圖 3-11 相同盤面示意圖.....24
圖 3-12 ZobristKey 示意圖.....25
圖 3-13 ZobristKey 拿掉相之後代表的盤面.....26
圖 3-14 ZobristKey 將相放到新位置之後代表的盤面.....26
圖 3-15 ZobristKey 將馬拿掉之後代表的盤面.....26
圖 3-16 盤面相同輪走方不同示意圖.....27
圖 3-17 盤面相同輪走方不同示意圖.....27
圖 3-18 節點存入 Hash Table 的函式虛擬碼.....29
圖 3-19 節點探測 Hash Table 的函式虛擬碼.....30
圖 3-20 Alpha-Beta 結合 Hash Table 的虛擬碼.....31
圖 3-21 空步示意圖.....32
圖 3-22 空步示意圖.....32
圖 3-23 Iterative Deepening 示意圖.....33
圖 3-24 子力評估範例 1.....34
圖 3-25 子力評估範例 2.....35
圖 3-26 威脅度範例 1.....36

圖 3-27 威脅度示意圖 36
圖 3-28 威脅度範例 2 37
圖 3-29 威脅度範例 3 37
圖 3-30 區域關係範例 37
圖 3-31 區域關係示意圖 38
圖 3-32 棋盤位置範例 39
圖 3-33 翻棋位置範例 1 40
圖 3-34 翻棋位置範例 2 40

表目錄

表 2-1 棋子編號圖.....6
表 3-1 Alpha-Beta 搜尋葉節點數量表.....21
表 3-2 棋盤位置分數.....39

第一章　緒論

第一節 暗棋規則及玩法介紹

暗棋，只使用中國象棋棋盤的一半，據傳發明者為"棋壇總司令"謝俠遜[1]。暗棋利用了中國象棋的棋子一面有文字，另一面空白這個特色翻轉棋子，產生了帶運氣成份的象棋變體，讓對奕時增加其趣味性。

暗棋的棋子不是放在交叉線上，而是放在格子上，共 4*8=32 格，剛好放下所有棋子。開始時，將所有棋子的背面向上，令人看不到棋子是什麼，然後洗亂棋子，再放到棋盤上。如圖 1-1：

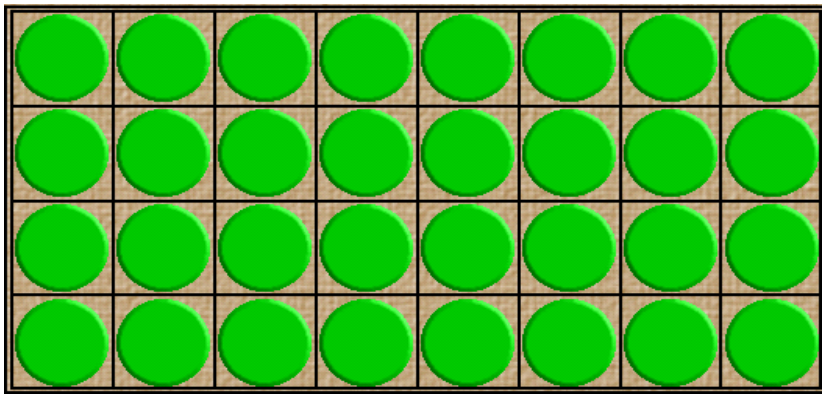


圖 1-1 暗棋的棋盤

未翻開的棋子稱為暗棋，翻開了的就叫明棋。

棋子樣子與數量完全與普通象棋一樣，如圖 1-2：

圖 1-2 暗棋的棋子

雙方輪流行走，每次可選擇翻開一枚暗棋、移動自己的一枚明棋或吃掉對手的一枚明棋。吃棋時的大小順序如下：

以下為紅方軍營的吃子關係：

帥可以吃的子為黑方的：將、士、象、車、馬、包。

仕可以吃的子為黑方的：士、象、車、馬、包、卒。

相可以吃的子為黑方的：象、車、馬、包、卒。

俥可以吃的子為黑方的：車、馬、包、卒。

傌可以吃的子為黑方的：馬、包、卒。

炮隔一子可以吃掉所有黑方的棋子。

兵可以吃掉黑方的：將、卒。

以下為黑方軍營的吃子關係：

將可以吃的子為紅方的：帥、仕、相、俥、傌、炮。

士可以吃的子為紅方的：仕、相、俥、傌、炮、兵。

象可以吃的子為紅方的：相、俥、傌、炮、兵。

車可以吃的子為紅方的：俥、傌、炮、兵。

馬可以吃的子為紅方的：傌、炮、兵。

包隔一子可以吃掉所有紅方的棋子。

卒可以吃的子為紅方的：帥、兵。

遊戲判斷勝負的方法為，如果紅方將黑方所有棋子吃光，那紅方則勝利，反之若黑方將紅方的子全部吃光，則黑方勝利。為了避免雙方都無法將對方子吃光

造成無限走棋使得遊戲永遠無法結束的狀況出現，當雙方連續無吃棋或翻棋的走步合計達到某個上限，則是為雙方和棋，一般來說，通常上限為二十五步或五十步。為了避免意志堅強的玩家，明知大勢已去卻仍抵死不放棄，徒然浪費時間，所以有長捉禁手這個規則存在。

長捉禁手：玩家(對手)將不能追擊對方(玩家)同一顆棋子連續 10 次，以避免一場棋玩了很久還結束不了的僵局，如圖 1-3：

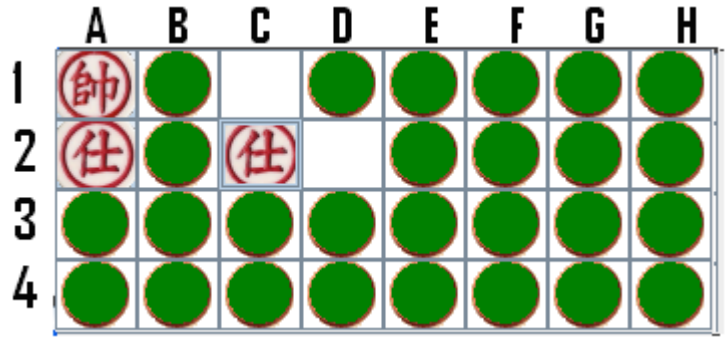


圖 1-3 長捉示意圖

此時輪黑方，黑方翻了 D1 翻到包，如圖 1-4：

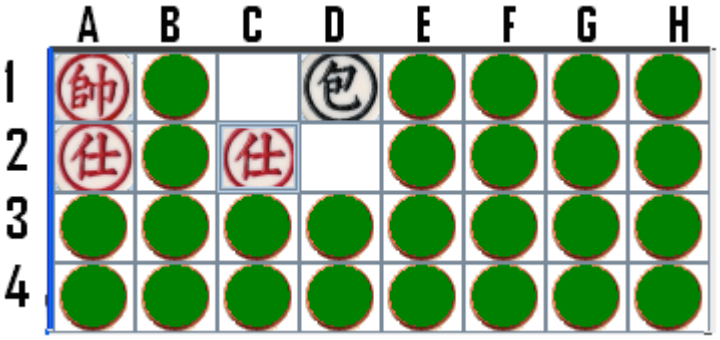


圖 1-4 長捉示意圖

此時如果紅方走 C2-C1 捉包，黑方走 D1-D2 逃包，紅方又走 C1-C2 捉包，黑方走 D2-D1 逃包，紅方再 C2-C1 捉包...， 紅方為了不讓帥或仕被包吃掉，一直長

捉黑包，造成雙方僵持不下的局面。所以，長捉在遊戲進行的過程中是不被允許的。故此例紅方不可一直長捉黑包，必須改走其它棋步（如翻棋），否則紅方視為輸棋。

第二節　論文概要

本論文共有四章，第一章首先介紹了暗棋與其基本規則和玩法，第二章則介紹了我們程式會用到的資料結構與走步產生方式，經過我們改良了的走步產生方式，減少了大量的判斷次數。第三章則大略講解了我們會用到的相關搜尋演算法與審局函數，其中我們的審局函數與原本謝曜安的審局函數有相當大的改變，除了吃子之外更考量到了位置關係與每顆子不同盤面的不同子力。第四章則作了初步的實戰測試與結論和未來研究方向 。

第二章　資料結構

第一節　棋盤-棋子映射結構

我們的暗棋棋盤結構和謝曜安[4]是使用一維陣列 Board[0]~ Board[31] 來記錄棋盤上每一格的棋子類型如圖 2-1：

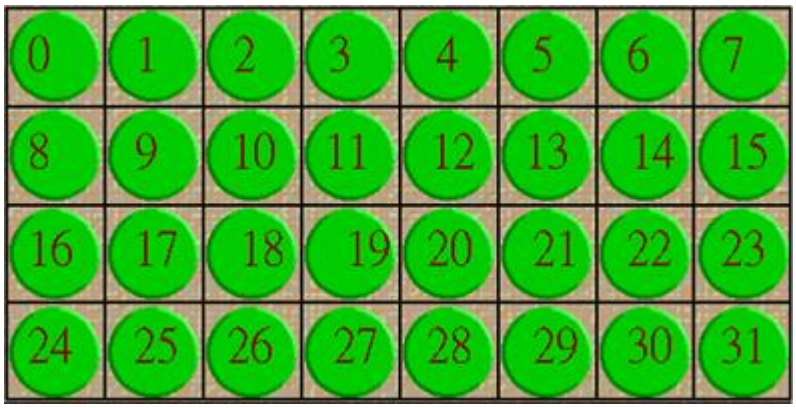


圖 2-1 一維陣列表示法

如果 Board[0]=8，代表棋盤最左上角的棋子為編號 8 的棋子，棋子編號如表 2-1，所以棋盤最左上角為紅傌，之所以沒有採用較為直觀的二維陣列原因是：一維陣列表示法擁有比較快的存取速度，因為高階語言中的二維陣列，經編譯程式處理之後，仍然是被轉換成一維的記憶體，中間多了一道索引值轉換處理的動作，因此一維陣列表示法會比二維陣列表示法要來得快[6]。

有了棋盤結構後，還需要一個棋子結構來記錄每個棋子位在棋盤上的哪個位置，這邊我們使用的是一維陣列 Position[1] ~Position[32]來識別 32 個棋子[4]，例如若編號 8 的紅傌位於棋盤最右上角，則 Position[8]=0，如果 Position[K]為 33 代表棋子 K 沒有在棋盤上或是尚未被翻開，例如若編號為 12 的紅相尚未翻開或

是被吃掉了，則 Position[12]=33。

表 2-1 棋子編號圖

	暗棋								
	0								
紅方	紅兵					紅炮		紅傌	
	1	2	3	4	5	6	7	8	9
	紅俥		紅相			紅仕		紅帥	
	10	11	12	13		14	15	16	
黑方	黑卒					黑包		黑馬	
	17	18	19	20	21	22	23	24	25
	黑車		黑象			黑士		黑將	
	26	27	28	29		30	31	32	
	空格								
	33								

棋盤 Board、棋子 Position 結構均可對一個局面進行完整的描述[4]。單用棋盤 Board 結構能方便地檢閱棋盤上任一位置的資訊，但要產生合法走步時，卻要對整個棋盤進行一次檢查，共 32 次的比對才能找到每個棋子所在位置，效率較

低；單用棋子 Position 結構能方便知道某棋子在棋盤上的位置，但要檢閱棋盤狀況，例：是否能吃子時，最壞情況下卻要對 16 個棋子掃描一次。為了充分利用兩者各自的優點，所以把這兩種方法結合起來形成所謂的棋盤-棋子映射結構，當棋子移動時同時對棋盤、棋子結構進行調整，以達到快速檢索、檢閱的目的[4]。例如圖 2-2：

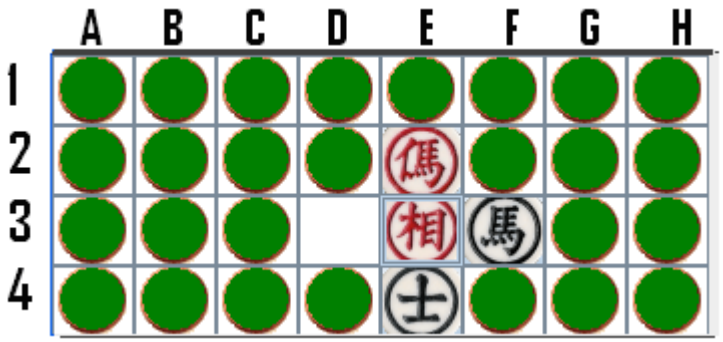


圖 2-2 board 與 position 示意圖

此例 Board[0]~ Board[31]陣列內容如下：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	33	12	24	0	0	0	0	0	0	30	0	0	0

Position[1] ~Position[32]其代表紅黑雙方的陣列內容如下：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
33	33	33	33	33	33	33	12	33	33	33	20	33	33	33	33

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	33	33	33	33	33	33	21	33	33	33	33	33	28	33	33

本來謝曜安的程式中只有使用 Position 的棋子結構陣列與 Board 的棋盤結構，在對雙方展開走步時，藉由 Position 確認我方哪些棋子在盤面上難免有些多餘的判斷，畢竟通常盤面上雙方 16 個棋子皆在盤面上的機會不高。所以我們多加了一個記錄雙方盤面上現有棋子的結構，以減少判斷盤面上有哪些棋子的動作，這裡我們使用的是一維陣列 current[1]~ current[32]來記錄雙方現在於盤面上的棋子，current[1]~current[16]為紅方，current[17]~current[32]為黑方，與記錄雙方現有棋子總數的結構 current_num[0]~ current_num[1]，current_num[0]為紅方現在盤面上的棋子總數，current_num[1]為黑方現在盤面上的棋子總數，再用一 current_index[34]陣列記錄盤面上每個棋子位於 current 結構中的位置，用於走步和回復走步用。若走步為吃子,則被吃方 current 陣列需更新，根據被吃子編號查詢 current_index 得到其於 current 中的位置，將他以那方最後一顆子取代，並且將那方的 current_num-1，回復走步則執行 current_num+1，且將被吃子根據 current_index 塞回 current 原位置中。如圖 2-3：

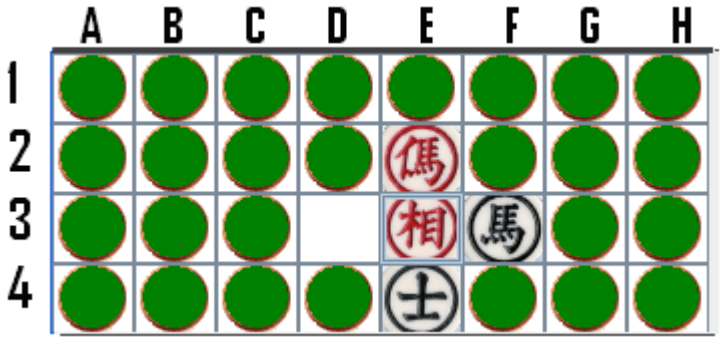


圖 2-3 current 示意圖

此例，current 中代表紅黑雙方盤面上棋子的 current[1]到 current[32]如下:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
8	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
24	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0

current_num[0]~[1]中的代表紅方盤面上棋子數的 current_num [0]為 2，代表黑方盤面上棋子數的 current_num [1]則為 2。

記錄雙方棋子位於 current 中位置的 current_index[1]~ [32]如下：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
33	33	33	33	33	33	33	1	33	33	33	2	33	33	33	33

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	33	33	33	33	33	33	17	33	33	33	33	33	18	33	33

,其中若值為 33 代表棋子不存在於盤面上。

若走了吃子走法，如圖 2-4：

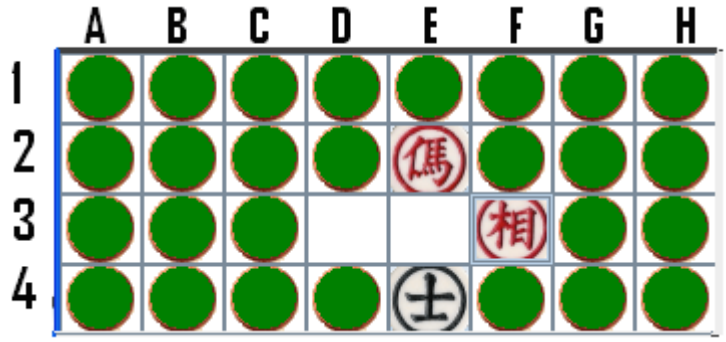
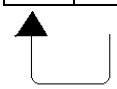
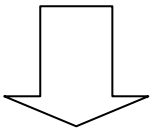


圖 2-4 current 更新示意圖

此例，紅相吃了黑馬，根據黑馬編號為 24 則查黑方 current_index 得知 current 黑馬的位置為 17，所以將 current[17]以 current 黑方最後一子 current[18]取代，黑方 current 更新過程如下：

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
24	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0



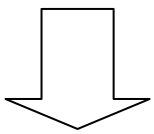


17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
30	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0

此時代表黑方盤面上棋子數的 current_num[1]由 2 遞減 1 為 1，這樣雖然 current 中有兩個代表黑士的 30，但是由於 current_ num [1]已經遞減為 1，所以最後一個 30 在產生走步時不會重覆讀取。在回復走步時，只要將被吃子根據 current_index 放回原來位於 current 中的位置，並將 current_num 遞增 1，即可回復到吃子前的狀態，回復過程如下：

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
30	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0

24



17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
24	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0

第二節　著法預處理

我們使用了 MoveTab[P][0]~ MoveTab[P][4] 二維陣列預先將位置 P 經由上、下、左、右移動一格後所能走到的位置儲存下來，其中 MoveTab[P][0] 代表位置 P 能走到的方向數量，而 MoveTab[P][1] ~ MoveTab[P][4] 則是記錄位置 P 走各方向後所到達的目的位置[4]，例如圖 2-5：

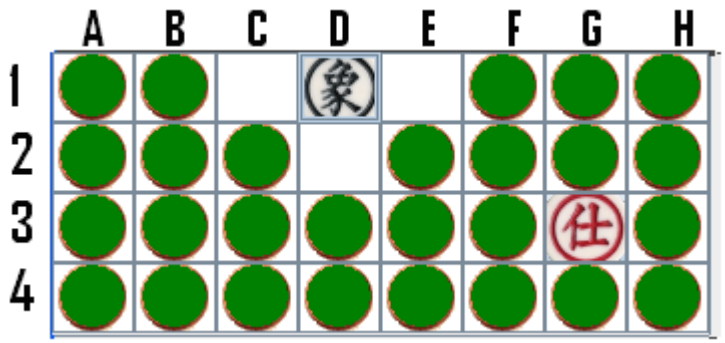


圖 2-5 MoveTab 示意圖

黑方 D1(位置 3)的象要怎麼知道有哪些方向可以走，它只要查 MoveTab[3][0]=3 發現有 3 個方向可以走，而依序存取 MoveTab[3][1] 的 2（左移一格後的目標位置）、MoveTab[3][2]的 4（右移一格後的目標位置）、MoveTab[3][3]的 11（下移一格後的目標位置），就可以得到 D1 所能走到的各方向的目標位置。這樣的做法會比每次都要上、下、左、右四次位移量加上還要判斷是否超出邊界要來的有效率。

除了預先處理著法可走到的方向外，我們還使用了 OrderTab[K1][K2] 來比較兩顆棋子的大小，其中 K1 代表來源棋子的編號，而 K2 代表目的棋子的編

號[4]。如果 OrderTab[K1[K2]的值是-1 的話，表示 K1 不能吃 K2；如果 OrderTab[K1][K2]的值是 0 的話表示 K1 為空格 K2 可走步到 K1 的位置；如果 OrderTab[K1][K2]的值是 1 的話表示 K1 可以吃 K2。因此欲知兵是否可吃將則查詢 Order_Tab[1][32]得值為 1，表示可以吃。欲知紅傌是否可吃象則查詢 Order_Tab[8][28]得值為-1，表是不可以吃。欲知兵可否往空格走，則查詢 Order_Tab[1][33]得值為 0，表示可以走(但不可以吃)。欲知兵可否往暗棋走，則查詢 Order_Tab[1][0]得值為-1，表示不可以吃。

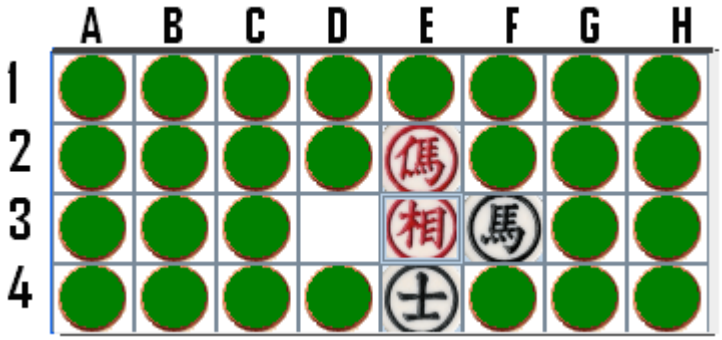


圖 2-6 OrderTab 示意圖

再來看圖 2-6 的例子，E3（位置 20）的紅相檢查 MoveTab[20][0]=4 會發現有 4 個方向可以走，分別是 E2（位置為 MoveTab[20][1]=12）、D3（位置為 MoveTab[20][2]=19）、F3（位置為 MoveTab[20][3]=21）、E4（位置為 MoveTab[20][4]=28）。其中 E3 紅相的棋子編號 12，而 E2 紅傌的棋子編號是 8，檢查 OrderTab[12][8] = -1，表示紅相不能吃紅傌，所以 E3-E2 不是一個合法步；而 D3 空格的棋子編號是 33，檢查 OrderTab[12][33] = 0，表示 E3 紅相可以走到空格，所以 E3-D3 是一個合法步；而 F3 黑馬的棋子編號是 24，檢查 OrderTab[12][24] = 0，表示紅相能吃黑馬，所以 E3-F3 是一個合法步；而 E4 黑士的棋子編號是 30，檢查 OrderTab[12][30] = -1，表示紅相不可以吃黑士，所以 E3-E4 不是一個合法步。這樣的做法會比使用一堆 if 指令來判斷是否可吃子、目的格是否為空格、目的格是否為暗棋要來得有效率。

第三節 著法產生流程

若輪走方欲產生所有合法走步，其步驟如下:

1. 首先查詢 current 表格，得到輪走方盤面上棋子
2. 根據 current 查詢 Position 表格，得到輪走方盤面上棋子位置
3. 根據其位置查詢 Mov_Tab 表格，查詢可走位置
4. 根據可走位置查詢 Board 表格查詢可走位置的棋種
5. 查到可走位置的棋種後，根據輪走方棋種與可走位置棋種查 Order_Tab 查看是否為合法走步，若是則產生走步，且優先產生吃子步

以圖 2-6 為例，若紅方要產生合法走步，首先去查詢 current[0]得到 2，代表紅方有兩顆棋子在棋盤上，所以對 current[1]~[16]中的前兩格掃描，得到我方盤面上的棋子為編號 8 的紅傌、編號 12 的紅相，所以先查詢 Position[8]得到位置為 12，再根據其位置 12 去查詢 Mov_Tab[12][0]為 4，代表有四個可以走到的位置，所以查詢 Mov_Tab[12][1]~ Mov_Tab[12][4]得到可走位置為 4、11、13、20，之後依據這四位置查詢 Board 得到可走到的位置棋種為 0、0、0、12 ，亦即棋種分別為暗棋、暗棋、暗棋、紅相。接著查詢 Order_Tab[8][0] 、Order_Tab[8][12]得到-1 與-1，故編號 8 的紅傌無合法走步。之後再如同上面做法去對編號 12 的紅相產生合法走步，我們共得到了紅相從位置 20 到 19 與紅相從位置 20 到 21 的兩個合法走步。

第三章　搜尋演算法

第一節 簡介

人類在下棋時，先手和後手會輪流猜測對方可能的走法，去思考因應的對策，先手方可能會思考若他下了這步棋，後手方可能會如何因應，以此方式判斷，先手方會想辦法下出後手方蒙受較多損失且我方得到較好利益的棋步。而暗棋程式同樣的也必須具備推算對方可能應對走法的能力，並且加以審慎評估所有可能的走法組合，然後下出最有利的一步棋。

棋類遊戲中通常會利用樹狀結構（又稱：對局樹或是遊戲樹）[1]將對奕的過程給表現出來，如圖 3-1 是井字遊戲中搜尋 2 層的對局樹[1]：

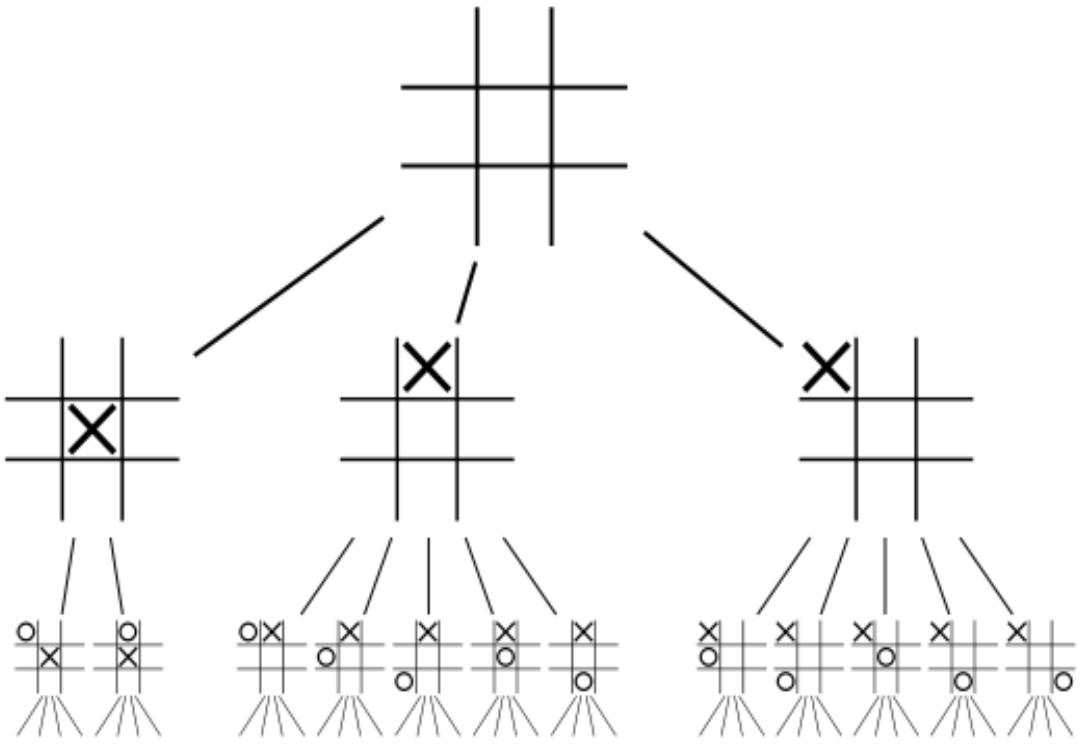


圖 3-1 井字遊戲對局樹範例

從圖 3-1 不難發現，對局樹的節點數隨著層數的增加呈指數倍的增加，如果一步棋有 k 種走法，而雙方共要走 n 步才能分出勝負的話，這樣一來總共有 k^n 種狀態需要搜尋，對於許多棋類遊戲來說，這樣的搜尋數量過於龐大，難以在合理的時間內搜尋完成，所以妥協的方法通常就是給有限層數下的搜索，或者是說只選擇某些走法去做展開卻希望仍然可以保留足夠的資訊。

當搜尋深度用完時會到達對局樹的葉節點，此時需要以審局函數來對此盤面做個評估，用來判斷從根節點到此葉節點的過程中對敵我雙方的好壞程度，將此分數依序往上傳給父節點，重複此過程直到整顆對局樹都搜尋完了，根節點保留了對先手方最有利的的分數，根據此分數據去走出對先手方最有利的一個走步。

第二節 Min-Max 搜尋演算法

一般來說，下棋時雙方都會走出使得盤面變成對自己最有利的棋步，而 Min-Max 搜尋法就是假設敵我雙方都選擇最佳走法時，一種以深度優先搜尋的演算法。它從根節點出發，透過著法產生器產生所有可能的走法後，以深度優先的方式向下進行節點的拜訪，當拜訪到給定深度後，在葉節點處使用審局函數判斷盤面好壞。

Min-Max 搜尋演算法在先手方走步時，會選擇能得到最大分數的走法。反之，在後手方走步時，則是選取得到最小分數的走法，因為後手方會希望先手方的利益為最小。Min-Max 搜尋演算法在對局樹不同層中，交互使用取最大值和最小值的走法，這也就是為什麼叫做 Min-Max 搜尋的原因。如圖 3-2：

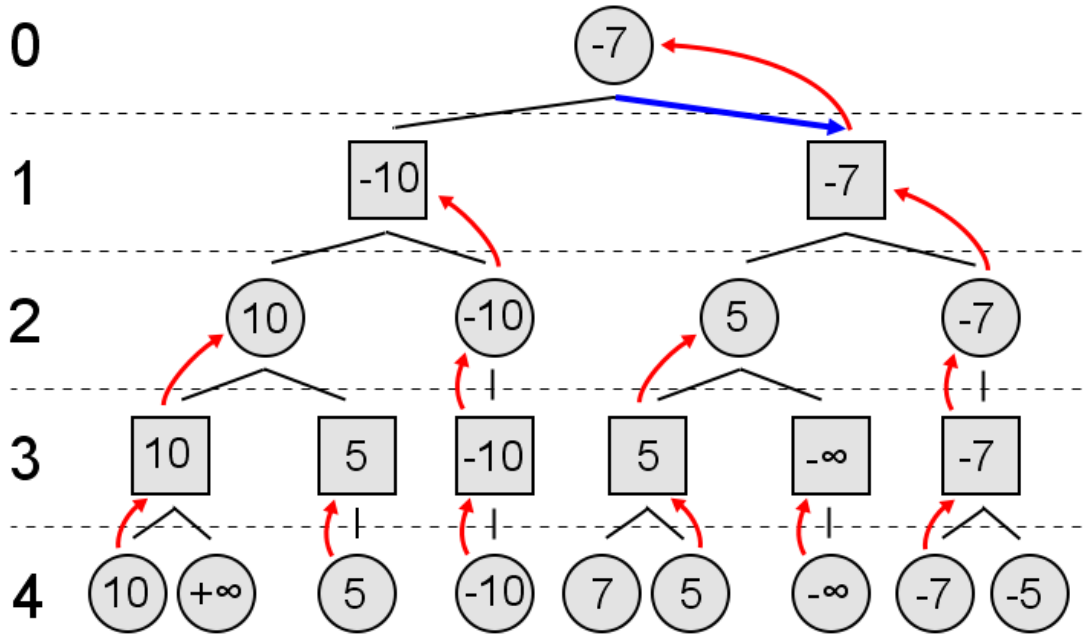


圖 3-2 Min-Max 示意圖

在遊戲樹中搜尋情況則如圖 3-3：

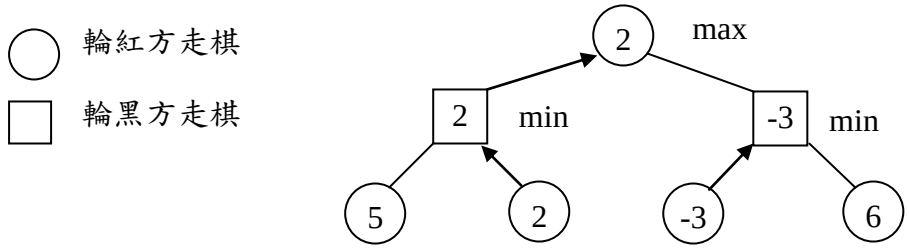


圖 3-3 棋類中 Min-Max 搜尋示意圖

假設紅黑雙方輪流各下一手棋之後共有 4 種可能盤面，而每個盤面根據審局函數會回傳不同的分數，最後根據每個節點輪紅方或黑方走步，分別選擇能夠得到最大或最小分數的走法，如此以深度優先的方式進行著，最後便能得到在根節點處對紅方而言最好的走法[6]。圖 3-4 是 Min-Max 搜尋演算法的虛擬碼：

```
int MinMax(int depth){
    int best,val;
    if(depth==0){
        return Evaluate();          //葉節點傳回審局函數的分數
                                    //總是以 root 方也就是我方的角度來評分
    }
    if(SideToMove()==My){           //輪到我方走棋
        best=-INFINITY;             //我方要取最大值，所以將 best 設成無限小
    }
    else{
        best=INFINITY;              //敵方要取最小值，所以將 best 設成無限大
    }
    GenerateLegalMoves();           //產生所有合理著法
    while(MoveLeft()){              //嘗試每一個走步
        MakeNextMove();             //走棋
        val=MinMax(depth-1);        //遞迴呼叫，層數減 1
        UnMakeMove();               //還原走步
        if(SideToMove()==My){
            if(val>best){           //我方要保留最大值
                best=val;
            }
        }
        else{
            if(val<best){           //敵方要保留最小值
                best=val;
            }
        }
    }
    return best;                    //將分數返回給父節點
}
```

圖 3-4 Min-Max 搜尋演算法虛擬碼

第三節 Nega-Max 搜尋演算法

第二節中所提到的 Min-Max 搜尋演算法總是需要檢查哪一方要取最大值，而哪一方又要取最小值，而執行不同的動作。在 1975 年 Knuth 和 Moore 提出了 Nega-Max 搜尋演算法[3]，使得不需要去判斷現在是哪一方而取最大值還是最小值，程式碼也因此變得簡潔優雅。如圖 3-5 是 Nega-Max 搜尋演算法的虛擬碼：

```
int NegaMax(int depth){
    int best=-INFINITY,val;
    if(depth==0){
        return Evaluate();          //葉節點傳回審局函數的分數
                                    //是以葉節點的角度來評分
    }
    GenerateLegalMoves();
    while(MovesLeft()){
        MakeNextMove();
        val= -NegaMax(depth-1);     //這裡要取負號
        UnMakeMove();
        if(val>best){
            best=val;
        }
    }
    return best;
}
```

圖 3-5 Nega-Max 搜尋演算法虛擬碼

從圖 3-5 的虛擬碼可以看出 Nega-Max 比 Min-Max 撰寫起來較為簡潔，而它的核心在於：藉由最大值取負號的行為取代了取最大或最小的動作。之所以能這樣做的原因是我們今天要解決的是屬於“零和”的棋類博奕，所謂“零和”指的就是說，在遊戲的任何一個時刻，一方獲得的利益就相當於另一方的損失，不會出現

雙贏的局面，雙方的利益所得加起來等於零，例如：西洋棋、象棋，還有我們這次研究的暗棋，都是屬於“零和”的棋類博奕[4]。所以，在算法的原理上 Nega-Max 和 Min-Max 是完全等效的，Nega-Max 僅僅是一種更好的表達形式，更有利於程式的撰寫。

第四節 Alpha-Beta 搜尋演算法

不論是在 Min-Max 或是 Nega-Max 搜尋演算法中，對於對局樹裡的每個節點的每一個所有可能著法都要走過一次，假設對局樹中每個節點平均有 k 個著法，如果要搜尋 n 層，就要拜訪大約 k^n 個節點，因此這樣做需要耗費相當多的時間，效率非常地差。

Brudno 在 1963 年提出了 Alpha-Beta 搜尋演算法[13]，它在大部份的情況下會比 Min-Max 有更佳的搜尋效率。而 Alpha-Beta 的核心思想是：在搜尋的過程中，發現無論如何都無法改變對方目前的最佳分數時，就可以提早放棄，不必浪費時間搜尋其它的著法。例如圖 3-6：

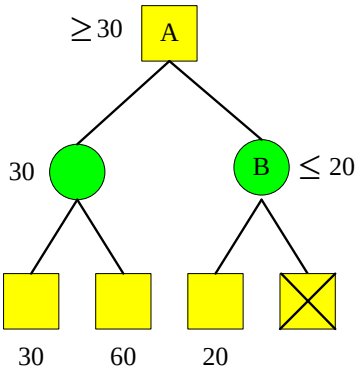


圖 3-6 Alpha-Beta 示意圖

圖 3-6 中，方形節點會從子節點中取最大值，圓形節點會從子節點中取最小值，而葉節點則會傳回以根節點為評分角度的審局函數值。其中節點 A，經由深度優

先搜尋左子樹，得到左子節點的值為 30，由於節點 A 會從其子節點中取最大值，所以 A 的值必不小於 30，換句話說，A 的右節點的值必須大於 30 ，才可能會被 A 選上。

當找尋過 B 的左子節點時，得到分數 20，因為 B 要取最小值，所以 20 便是 B 的最大可能值。然而，B 之值必須大於 30 才會被 A 選上，且 B 剩下的子節點無論分數如何，都不可能使 B 的值大於 30，所以就不用往下搜尋剩下的節點了。

Alpha-Beta 搜尋演算法必須使用兩個參數，分別是 α 和 β。其中 α 是記錄最大層節點目前的最大值，而 β 是記錄最小層節點的最小值，兩個參數以傳值(call by value) 的方式傳遞給下層的子樹。如果在取最大值的時候，發現了一個大於等於 β 的值，就不用再對其它分枝進行搜尋，這就是所謂的 β 截斷；同理，在取最小值的時候，發現了一個小於等於 α 的值，也不用再對其它分枝進行搜尋，這就是所謂的 α 截斷。例如圖 3-7 是 Alpha-Beta 另一個比較複雜的範例：

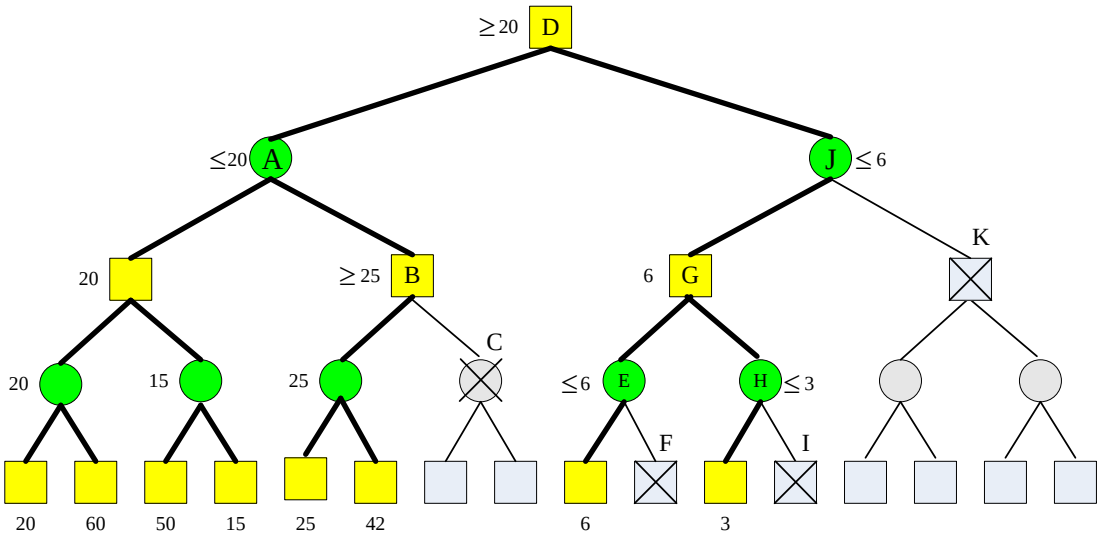


圖 3-7 Alpha-Beta 示意圖

圖 3-7 是 Alpha-Beta 以深度優先從左至右拜訪的樹狀圖，其節點截斷順序如下：

1. B 節點取值 25 的時候，25 ≧ 20，造成 C 節點的截斷（β 截斷）。

2. E 節點取值 6 的時候，6 ≦ 20，造成 F 節點的截斷（α 截斷）。

3. H 節點取值 3 的時候，3 ≦ 6，造成 I 節點的截斷（α 截斷）。

4. J 節點取值 6 的時候，6 ≦ 20，造成 K 節點的截斷（α 截斷）。

很顯然地，Alpha-Beta 搜尋演算法的搜尋效率與走法的排列順序有極密切的關係。如果總是先嘗試壞的走法的話，那最終將會與 Min-Max 搜尋演算法一樣，完全沒有任何截斷的機會。倘若能將好的走法排在前面優先嘗試的話，就可以儘早發生截斷，省掉對其它節點不必要的搜尋。例如圖 3-7 中，總共有 31 個節點，最終只搜尋了 19 個節點，省了 12 個節點。在最佳走法搜尋順序時，Knuth 和 Moore 證明了 Alpha-Beta 搜尋演算法最少必須搜尋的節點數目[3]為：

$$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$$

其中 b 為分支度，n 為深度。其最好與最壞的搜索情況如表 3-1：

表 3-1 Alpha-Beta 搜尋葉節點數量表

	在深度為 n 與 b = 40 時葉節點的數量表	
深度	**最差情況**	**最好情況**
n	b^n	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

圖 3-8 為 Alpha-Beta 搜尋演算法的虛擬碼：

```
int alphaBetaMax( int alpha, int beta, int depthleft ) {
    if ( depthleft == 0 ) return evaluate();
    for ( all moves) {
        score = alphaBetaMin( alpha, beta, depthleft - 1 );
        if( score >= beta )
            return beta;      // beta-截斷
        if( score > alpha )
            alpha = score; // alpha 為取最大值
    }
    return alpha;
}

int alphaBetaMin( int alpha, int beta, int depthleft ) {
    if ( depthleft == 0 ) return -evaluate();
    for ( all moves) {
        score = alphaBetaMax( alpha, beta, depthleft - 1 );
        if( score <= alpha )
            return alpha; // alpha-截斷
        if( score < beta )
            beta = score; // beta 為取最小值
    }
    return beta;
}
```

圖 3-8 Alpha-Beta 搜尋演算法的虛擬碼

圖 3-9 是以 Nega-Max 形式來表達 Alpha-Beta 搜尋演算法的虛擬碼：

```
int AlphaBeta(int depth,int alpha,int beta){
    int val;
    if(depth==0){
        return Evaluate();
    }
    GenerateLegalMoves();
    while(MovesLeft()){
        MakeNextMove();
        val= -AlphaBeta(depth-1,-beta,-alpha); //alpha 和 beta 順序對調
                                               //且加上負值
        UnMakeMove();
        if(val>=beta){
            return beta;                       //beta 截斷
        }
        if(val>alpha){
            alpha=val;
        }
    }
    return alpha;
}
```

圖 3-9 Nega-Max 形式的 Alpha-Beta 搜尋演算法虛擬碼

以 Nega-Max 形式寫成的 Alpha-Beta 搜尋演算法，讓程式碼更簡潔有力[4]。

第五節 Transposition Table

盤面狀態是完全一致但是經過不同走法順序所到達的情況，在對局樹的搜尋過程中有相當大可能出現，如圖 3-10：

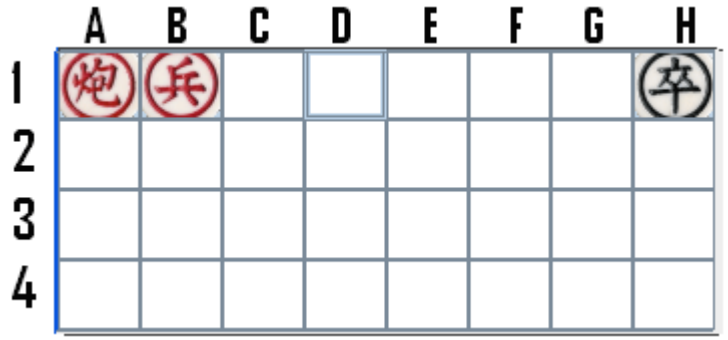


圖 3-10 相同盤面示意圖

不管是走路徑：H2-G2，B1-B2，G2-G3，A1-A2，還是走路徑：H2-H3，B1-B2，H3-G3，A1-A2，都可以到達如圖 3-11 的盤面：

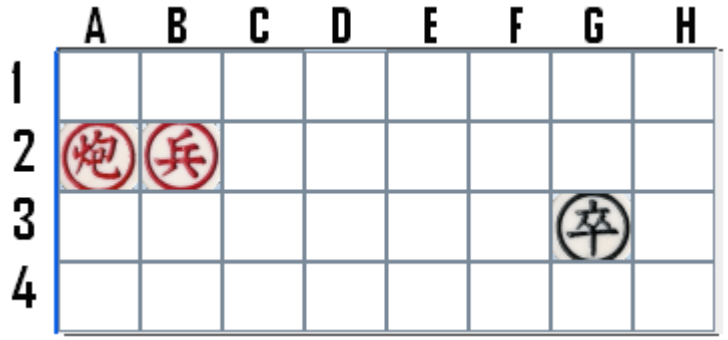


圖 3-11 相同盤面示意圖

從以上的例子可以知道，如果之前有某個盤面節點已經搜尋過的話，當再次遇到同樣狀態盤面節點時，就可以套用之前搜尋的結果，省下再重複搜尋的時間。而 Transposition Table（又稱 Hash Table）[11]，則是一個可以將已搜尋過的節點資訊記錄起來的雜湊表。節點資訊通常包括：盤面 key 值、節點搜尋深度、節點分數、節點類型…等。一般使用 Zobrist Hash 的方式來進行盤面 key 值的生成，以達到快速檢測當前節點是否已經搜尋過的目的[4]。

Zobrish Hash 方法是在搜尋之前，預先產生大隨機數的二維陣列 Zobrist[棋

子類型][棋子位置]，我們的大隨機數是使用 32 位元的無號整數，有了 Zobrist[棋子類型][棋子位置] 這樣的二維陣列以後，當前盤面的 key 值，便是所有棋盤上的棋子相對應到 Zobrist[棋子類型][棋子位置]二維陣列做 xor 運算（以⊕表示）後的和。這樣在有著法產生移動時，並不需要重新計算盤面的 key 值，只需要將當前 key 值（假設叫：ZobristKey）做以下步驟[4]：

1. ZobristKey⊕=Zobrist[移動棋子類型][移動棋子原位置]，其意義在於消除移動棋子在原位置中對 key 值的影響，現在的 key 值代表的盤面即為將移動棋子拿掉的盤面。
2. ZobristKey⊕=Zobrist[移動棋子類型][移動棋子新位置]，其意義將移動棋子在移動後位置中對 key 值產生更動，現在的 key 值代表的盤面即為將移動棋子拿掉後並把移動棋子放上新位置後的盤面。
3. 如果著法是吃子著法的話，需要再做：ZobristKey⊕=Zobrist[被吃棋子類型][被吃棋子位置] ，其意義在於消除被吃棋子在原位置中對 key 值的影響現在的 key 值代表的盤面即為將移動棋子拿掉後並把移動棋子放上新位置後且將被吃棋子拿掉的盤面。

以圖 3-12 為例：

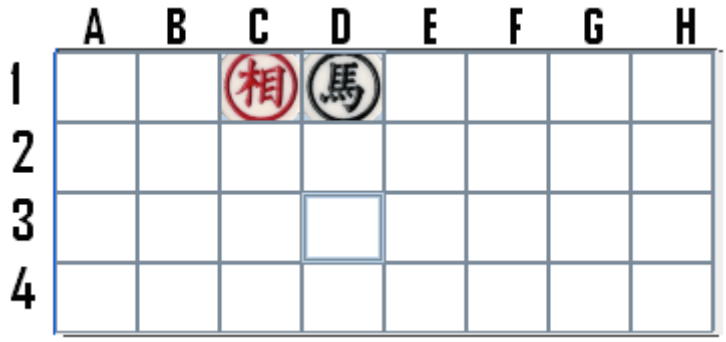


圖 3-12 ZobristKey 示意圖

初始的盤面 key 值為 ZobristKey=Zobrist[相][2]⊕zobrist[馬][3]。

則步驟 1 做完後 ZobristKey=ZobristKey⊕Zobrist[相][2]。其代表盤面如圖 3-13:

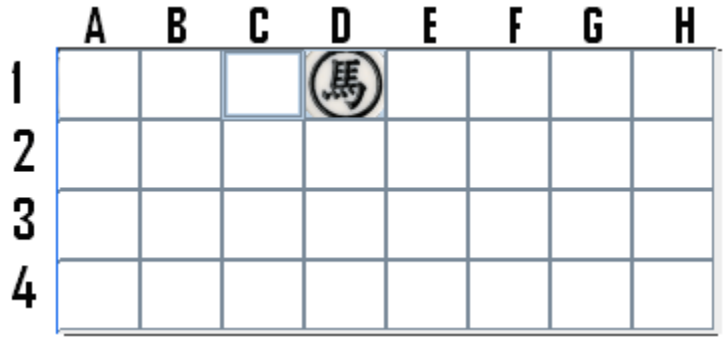


圖 3-13 ZobristKey 拿掉相之後代表的盤面

步驟 2 做完後 ZobristKey=ZobristKey⊕Zobrist[相][3],此時 ZobristKey 值代表的盤面相和馬皆在同一個位置上。其代表盤面如圖 3-14:

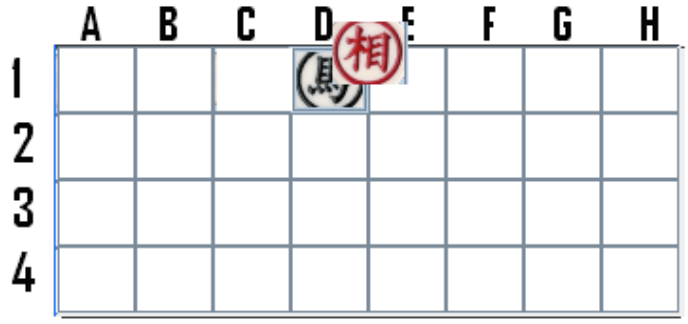


圖 3-14 ZobristKey 將相放到新位置之後代表的盤面

步驟 3 做完後 ZobristKey=ZobristKey⊕Zobrist[馬][3],此時 ZobristKey 值代表的盤面相即為將馬從盤面上移除。其代表盤面如圖 3-15:

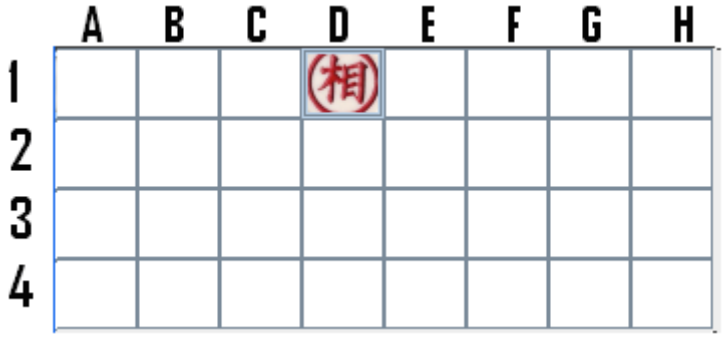


圖 3-15 ZobristKey 將馬拿掉之後代表的盤面

我們使用了一個大隨機數 ZobristPlayer 來代表輪走方，用來避免相同 key 值盤面相同下輪走方卻不同的情況，每走一步就做 ZobristKey⊕=ZobristPlayer 運算，最後的 ZobristKey 才是真正代表盤面且加上輪走方訊息的 key 值。我們以如圖 3-16 輪紅方走的盤面當做不加 ZobristPlayer xor 運算導致資訊引用錯誤的例子，此時的盤面 key 值 ZobristKey = Zobrist[卒][0]⊕Zobrist[兵][1]⊕Zobrist[炮][2] ⊕ Zobrist[卒][7]。

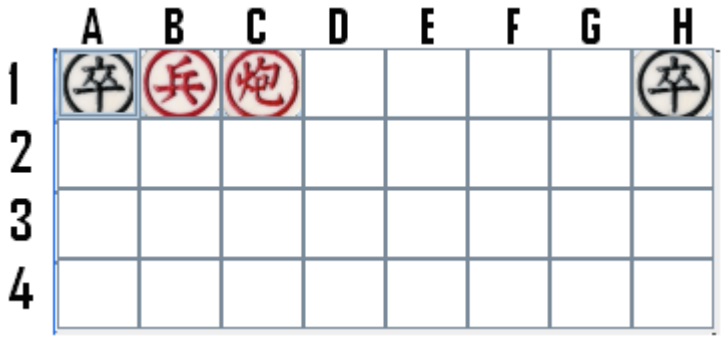


圖 3-16 盤面相同輪走方不同示意圖

紅方走了 C1-A1 吃卒後，如圖 3-17，此時輪黑方走，此時的盤面 key 值 ZobristKey = ZobristKey ⊕ Zobrist[炮][2] ⊕ Zobrist[炮][0] ⊕ Zobrist[卒][0] 。

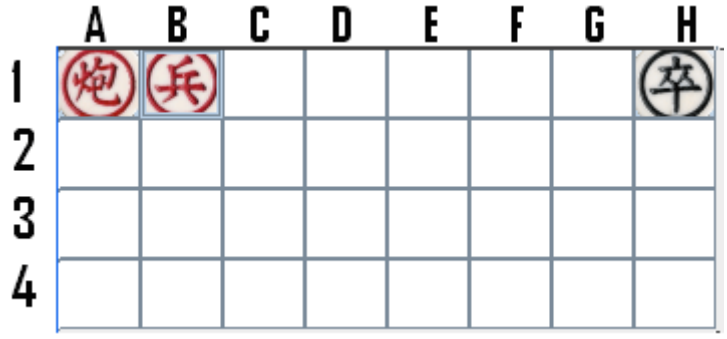


圖 3-17 盤面相同輪走方不同示意圖

如果圖 3-16 紅、黑雙方依路徑：B1-A1 吃卒，H1-H2，C1-C2，H2-H1，C2-B2，

H1-H2，B2-A2，H2-H1，A1-B1，H1-H2，A2-A1，H2-H1，

ZobristKey = ZobristKey ⊕

（B1-A1 吃卒）Zobrist[兵][1] ⊕ Zobrist[兵][0] ⊕ Zobrist[卒][0] ⊕

（H1-H2）Zobrist[卒][7]⊕Zobrist[卒][15] ⊕

（C1-C2）Zobrist[炮][2]⊕Zobrist[炮][6] ⊕

（H2-H1）Zobrist[卒][15]⊕Zobrist[卒][7] ⊕

（C2-B2）Zobrist[炮][6]⊕Zobrist[炮][9] ⊕

（H1-H2）Zobrist[卒][7]⊕Zobrist[卒][15] ⊕

（B2-A2）Zobrist[炮][9]⊕Zobrist[炮][4] ⊕

（H2-H1）Zobrist[卒][15]⊕Zobrist[卒][7] ⊕

（A1-B1）Zobrist[兵][0]⊕Zobrist[兵][1] ⊕

（H1-H2）Zobrist[卒][7]⊕Zobrist[卒][15] ⊕

(A2-A1) Zobrist[炮][4]⊕Zobrist[炮][0] ⊕

（H2-H1）Zobrist[卒][15]⊕Zobrist[卒][7]。

由於同值作偶數次 xor 原值不變，所以只有留下奇數次的的不同值 xor 則 ZobristKey= ZobristKey ⊕ Zobrist[炮][2] ⊕ Zobrist[炮][0] ⊕ Zobrist[卒][0]。也可以到達如圖 3-17 的盤面，但此時是輪紅走，而盤面 ZobristKey 值仍舊和輪黑走一樣。所以如果只用棋盤上所有棋子相對應的 Zobris t[棋子類型][棋子位置]做 xor 後的總和，這時就會發生盤面相同但輪走方不同卻擁有相同 key 值的情況，而造成搜尋時節點資訊的誤用。如果有在每走一步做 ZobristKey⊕=ZobristPlayer 運算，以圖 3-17 為例，輪紅方走時 ZobristPlayer 必然跟 ZobristKey 做了偶數次 xor 運算，而輪黑方走時 ZobristPlayer 必然跟 ZobristKey 做了奇數次 xor 運算，所以即使原來的 ZobristKey 相同，也會因為跟 ZobristPlayer 做了奇、偶次數 xor 運算的關係，而讓最後的 ZobristKey 值不同。

我們的 Hash_Table_Size 為 2^{20}，其中 slot 位置的算法為：slot = 盤面 key 值 %Hash_Table_Size[4]。其中%表示求餘數的運算。將 Hash_Table_Size 設為 2^N 這

個常量，要得到除以 2^N 後的餘數，只要將盤面 key 值& (Hash_Table_Size – 1) 即可，電腦做&運算（AND）的速度是非常快的。

由於實際記憶體空間有限，所以 Hash Table 的空間不可能無限大，因此我們必須對所產生的節點是否存入 Hash Table 做些限制，避免一些無用的節點資訊佔據了 Hash Table 的空間，而這些無用的節點通常是一些距離根節點很遠的節點，因為它們被重複搜尋的機率很低。當盤面值對應到同一個 slot 位置時，我們採用“深度優先覆蓋”的策略，所謂的“深度優先覆蓋”即待存入 Hash Table 的節點的深度必須大於或等於已存入 Hash Table 節點的深度才覆蓋。據研究表示：深度優先覆蓋的效果要比始終覆蓋或完全不覆蓋要好得多[2]，且程式撰寫起來相當簡易。如圖 3-18 是節點資訊存入 Hash Table 的函式虛擬碼：

```
void RecordHash(int depth,int val,int hashf){
    HashStruct *p = &Hash_Table[ZobristKey & (Hash_Table_Size-1)];
    if(depth>=p->depth){          //深度優先覆蓋，存入節點相關資訊
        p->key=ZobristKey;        //存入盤面 key 值
        p->depth=depth;           //存入節點深度
        p->val=val;               //存入節點分數
        p->hashf=hashf;           //存入節點類型
    }
```

圖 3-18 節點存入 Hash Table 的函式虛擬碼

由於 Alpha-Beta 搜尋演算法的過程中，任何節點的分數都是下列三種情況之一：

（一）節點分數 ≧ beta，即所謂的 beta 節點。

（二）節點分數 ≦ alpha，即所謂的 alpha 節點。

（三）alpha < 節點分數 < beta，即所謂的 PV（Principal Variation）節點。

通常來說只有 PV 節點的分數，才可以當作是節點的準確值存入 Hash Table

，其餘的 beta 節點、alpha 節點的分數只能表示是節點分數的一個邊界而已。但存入這樣的邊界分數，仍有助於我們下次搜尋到同樣盤面節點的時候，進行剪裁的動作[4]。所以在上面的虛擬碼中 Hash Table 裡最後還存了節點類型。

當目前節點探測 Hash Table 成功時，如果已存入的節點類型是 PV 的話，則直接返回節點分數，如果已存入節點類型是 beta 或 alpha 的話，需要進行邊界值的比較，決定是否能截斷。如圖 3-19 是節點探測 Hash Table 的函式虛擬碼：

```
int ProbeHash(int depth,int alpha,int beta){
    HashStruct *p = &Hash_Table[ZobristKey & (Hash_Table_Size-1)];
    if(ZobristKey==p->key&&p->depth>=depth){          //探測成功
        if(p->hashf==PV_FLAG){
            return p->val;
        }
        if(p->hashf==BETA_FLAG&&p->val>=beta){   //與上界 beta 比較
            return beta;
        }
        if(p->hashf==ALPHA_FLAG&&p->val<=alpha){//與下界 alpha 比較
            return alpha;
        }
    }
    return valUNKNOWN;                    //探測失敗或是截斷失敗
}
```

圖 3-19 節點探測 Hash Table 的函式虛擬碼

從上面的虛擬碼可以看出，所謂的探測成功，除了比較當前的盤面值與 Hash Table 的盤面值是否一樣以外，還要比較 Hash Table 的節點深度是否大於或等於當前節點的深度。這是因為 Hash Table 中的節點深度如果比當前節點深度淺的話，所存的節點資訊還不足以讓當前節點拿來使用，因為它的搜尋深度還不夠深。只有在 Hash Table 的節點深度大於或等於當前節點深度的時候，我們才能拿裡面的資訊來利用。圖 3-20 是 Alpha-Beta 搜尋結合 Hash Table 的虛擬碼：

```
int AlphaBeta(int depth,int alpha,int beta){
    int val,foundPV=0;
    if(depth==0)        return Evaluate();                    //回傳審局函數分數
    if((val==ProbeHash(depth,alpha,beta))!=valUNKNOWN){ //探測 Hash Table
        return val;
    }
    GenerateLegalMoves();
    while(MovesLeft()){
        MakeNextMove();
        val= -AlphaBeta(depth-1,-beta,-alpha);
        UnMakeMove();
        if(val>=beta){
            RecordHash(depth,beta,BETA_FLAG);          //存入 beta 節點
            return beta;
        }
        if(val>alpha){
            foundPV=1;                                 //有找到 PV 節點
            alpha=val;
        }
    }
    if(foundPV)   RecordHash(depth,alpha,PV_FLAG);     //存入 PV 節點
    else  RecordHash(depth,alpha,ALPHA_FLAG);          //存入 alpha 節點
    return alpha;
}
```

圖 3-20 Alpha-Beta 結合 Hash Table 的虛擬碼

第六節允許空步

如果盤面上還有暗棋未翻，且走的每一步明棋都不會得到好的結果時，強迫走明棋是不合理的，因為在現實大部分的狀況中，玩家通常會選擇去做翻暗棋的動作，這個作法是常用於電腦象棋中的空步搜尋[14]。例如圖 3-21，輪紅方走：

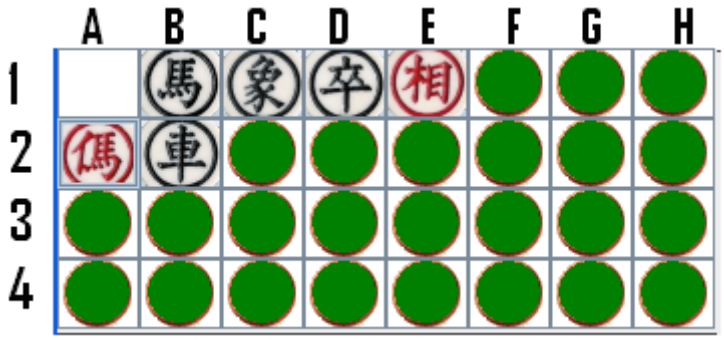


圖 3-21 空步示意圖

紅方走了 A2-A1 後，黑方走 B1-A1 吃紅馬後，如圖 3-22，輪紅方走：

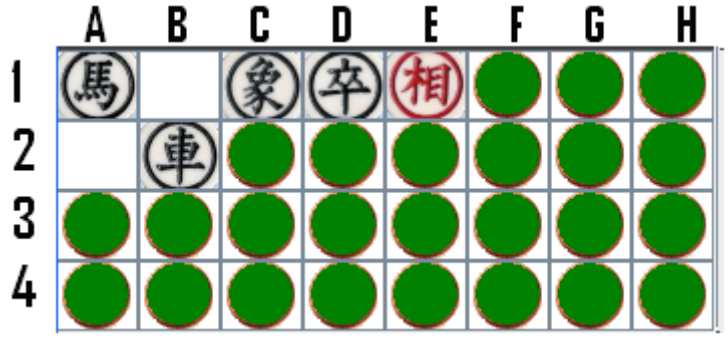


圖 3-22 空步示意圖

此時剩 E1-D1 相吃卒可以走，但吃了卒後，紅方可以走 C1-D1 象吃相，則紅方蒙受了不必要的損失。一般玩家這種情況下更傾向於翻暗棋，但由於在搜尋樹中若真的以翻各種棋來模擬，則搜尋樹的分枝度過於龐大，所以我們改以不走步來模擬翻棋的效果。所以若還有暗棋未翻且如果所走的每步明棋都會更糟時，我們會以允許空步模擬走棋之後作了翻棋的動作，讓審局函數可以反應出更真實的狀況。

第七節 Iterative Deepening

在 Alpha-Beta 搜尋演算法中，走步好壞的順序對於截斷與否是很重要的。通

常三層 Alpha-Beta 搜尋所產生的好步，在四層 Alpha-Beta 搜尋時也不會差到哪裡，所以我們使用了逐層加深(Iterative Deepening)[9]來幫助我們對走步做排序，例如三層 Alpha-Beta 搜尋完之後,每個走步會有個回傳值,將走步依其回傳值好壞做排序給四層的 Alpha-Beta 去搜尋,藉由走步的排序,可以提高截斷率。

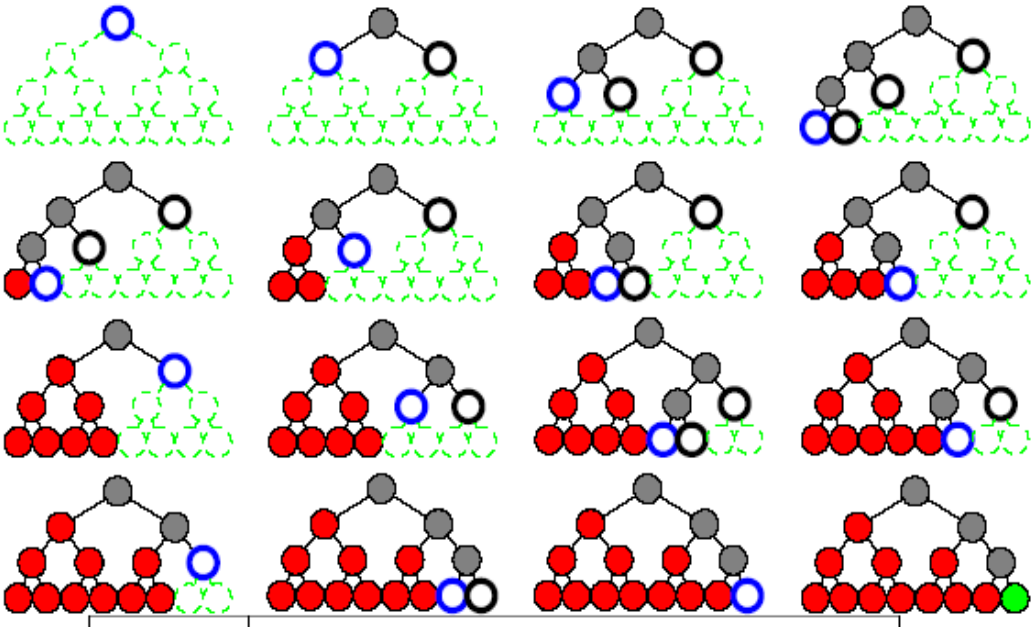


圖 3-23 Iterative Deepening 示意圖

如圖 3-23 若用逐層加深的搜尋方式，所需搜尋的節點數量，表面上似乎比直接用深度優先搜尋還多得多。然而，利用逐層加深的技巧來作搜尋，其所需找過的節點數量比直接用深度優先搜尋只多了 11%左右[3]，並且運用逐層加深的搜尋技巧，可以使較淺層搜尋中所得到的資訊，用來在較深層的搜尋中截斷更多的節點。

第八節 利用食物鏈的關係與威脅度設計審局函數

一般而言，帥的價值永遠比兵還大，讓我們來考量一個極端的情況，如圖

3-24，帥的價值應該趨近於零，因為盤面上對方的子沒有一顆是它能夠吃得下去的。為解決此問題所以我們在搜尋一開始時，會針對盤面尚存的子（包括明棋和未翻之棋），去計算每顆棋子還能夠吃多少顆子，且吃的子價值多少，將之加總起來再乘以那個棋子本身的權重，最後所有尚存之子再加上一個共同的基本分。具體方法如下：

（一） 將吃子關係分為兩類

1. 無法反擊的，如兵攻擊將。
2. 可以反擊的，如仕攻擊士。

（二） 計算每顆棋子的基本價值：

1. 每顆棋子基本價值為 1+4×(無法反擊的子數)+1×(可以反擊的子數)
2. 炮(包)基本價值則為 4×(我方棋子數)+1×(對方棋子數)。

（三） 每顆棋子的子力則為能吃到的所有棋子基本價值總和。

（四） 根據每個棋子棋種乘上它的棋種權重分。

（五） 炮(包)子力則為紅仕(黑士)×(4/15)。

（六） 每顆子最後再加上基本分數，我們基本分數是給 50。

例如在圖 3-24 的盤面下，由於帥已經沒有任何子可以吃了，所以將之加總之後乘以本身帥的棋子權重仍然等於零，再加上共同基本分，計算出來帥只有基本分的價值，但是兵還有卒可以吃，所以就算兵的棋子權重比較小，加總起來乘以棋子權重權重價值再加上共同基本分後就會會大於帥了。

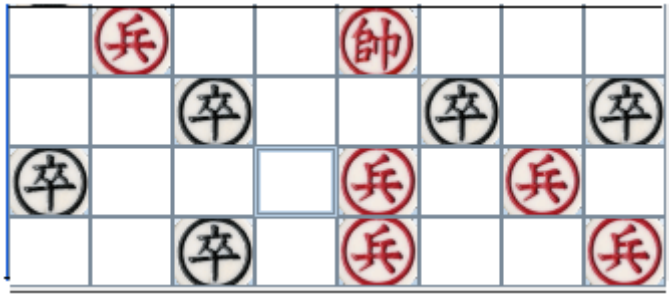


圖 3-24 子力評估範例 1

另一個例子如圖 3-25，黑方盤面上的黑士。能夠吃的子明顯比紅方多了不少，所以加總起來黑士的價值會比紅仕高，所以如果可以的話紅方會以一顆紅仕換對方一顆黑士。由於我們每個棋子都有給本身權重，所以就算黑士吃的總子力比紅帥的吃的總子力價值還多，但由於帥本身的棋子權重，所以幾乎不會發生紅方以紅帥換黑方黑士的情況。所以我們的審局函數會根據以上計算出來的子力價值，去計算紅黑雙方在搜尋路徑中共吃了哪些子，以此作為審局初步依據。

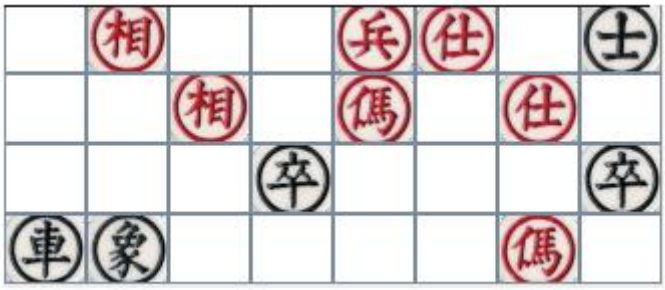


圖 3-25 子力評估範例 2

但若只有考量到吃子的話，在搜尋深度不夠深的情況下，難免有所不足，所以我們加了一個威脅度的考量，考慮圖 3-26 的盤面。則我們將盤面轉換成圖 3-27 的表示型態，其中箭頭代表吃子的關係，例如將有箭頭指到相，代表「將」可以吃相，「兵」有箭頭指向將，代表「兵」可以吃「將」，其中的 dk 則代表了兩顆子之間的距離*上距離權重，例如「將」指到相有個 d3 的距離，「兵」指到「將」有個 d2 的距離。所以例如「兵」走在距離 d2 的情況下可以威脅到「將」，紅方就得到一個（「將」的子力）/d2 的分數，「相」會被「將」在距離 d3 的情況下威脅，所以紅方會失去一個（「相」的子力）/d3 的分數。

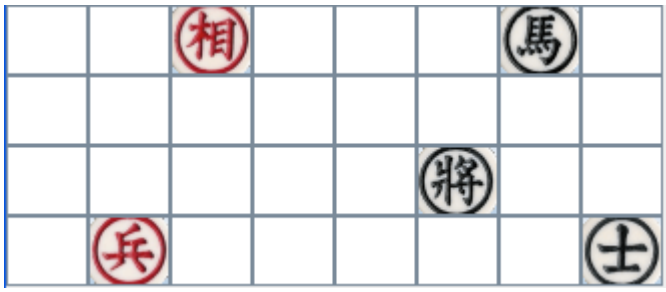


圖 3-26 威脅度範例 1

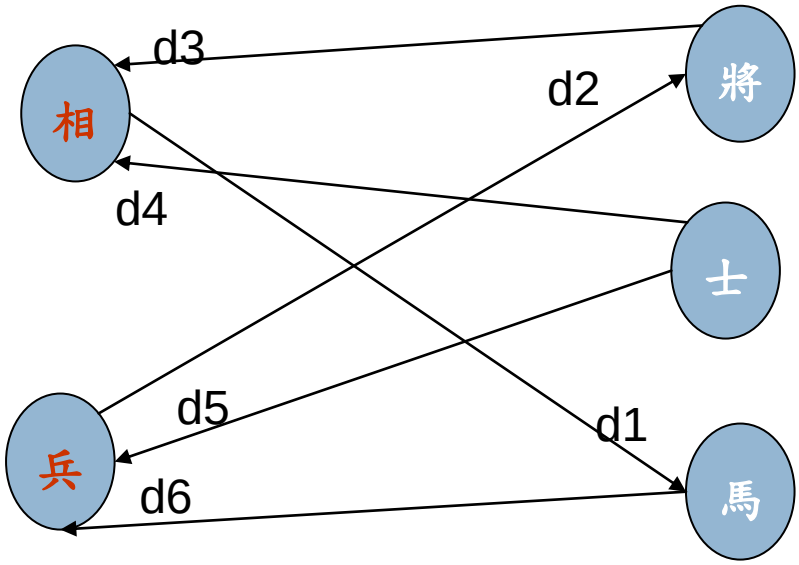


圖 3-27 威脅度示意圖

但若只以距離來考量威脅度，則會發生盲目追擊或盲目竄逃的情況，例如圖 3-28，若單以距離計算威脅度，黑方「象」往「兵」靠近一步，則威脅度更大，但是這樣下去「象」永遠吃不到兵」，在能吃對方的情況下，只有距離為偶數步的情況才有威脅性，所以在能吃對方時但距離為奇數步時，其威脅度產生的分數仍然為零，所以圖 3-28 的情況下象往前一走會造成距離為奇數步，此時「兵」不管怎麼走，若只考量距離的情況下「象」刻意會往「兵」靠近，但是這樣下去永遠吃不到「兵」，而且會造成長捉的違規情況，所以若考量到奇數步和偶數步的情況下，此時黑方會走「士」，以維持「象」對「兵」的威脅度。

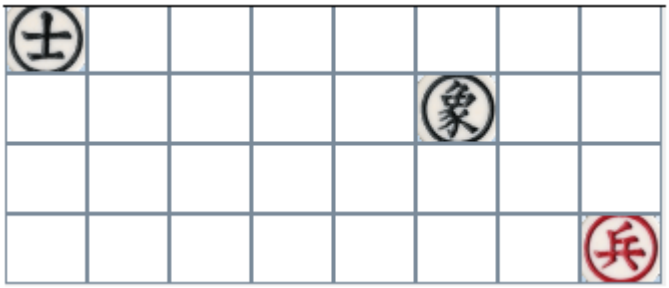


圖 3-28 威脅度範例 2

若是在可能被吃的情況下，則是距離為偶數步才為安全，例如圖 3-29 的情況下，由於紅兵已經與黑象保持了偶數步距離，若一味往外逃，反而會持續被黑象所威脅，所以我們此時應去移動紅俥。

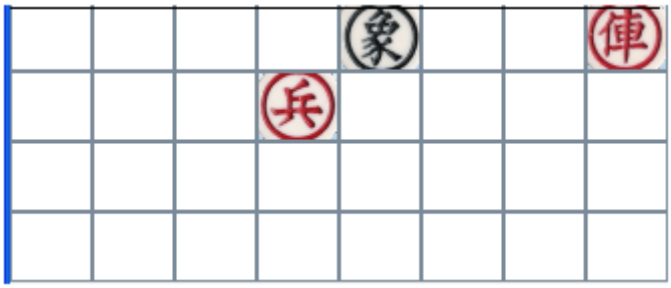


圖 3-29 威脅度範例 3

審局函數的分數仍然依吃子為主要考量依據，最後再加上威脅度來當作輔佐，以用在搜尋深度不足的情況下來更精確的審視盤面。

圖 3-30 區域關係範例

不過有時候某些不同區域的棋子在根據目前的盤面下搜尋時並不會有威脅對方的情形。例如圖 3-30 所示，此盤面被暗棋分成了四大塊如圖 3-31 所示，不同顏色表不同，區域黑色表暗棋，除了紅炮黑包之外並無法產生威脅。所以只有在同個區域中才會有威脅度的考量。

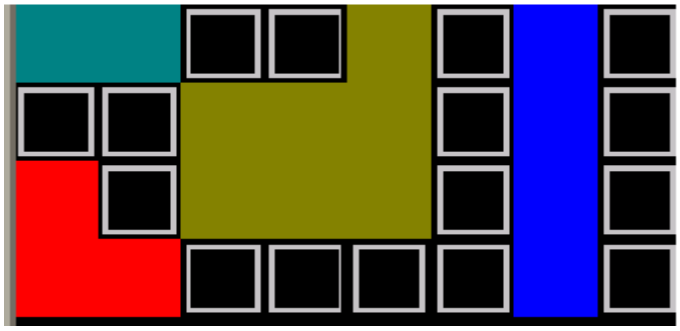


圖 3-31 區域關係示意圖

我們會對盤面上不為暗棋的點輪流作擴散的動作，在擴散過程中經過的位置標上區域編號，在同一輪擴散中經過的位置標上的區域編號皆相同，當擴散到沒有未標區域編號或都暗棋時，這輪擴散結束，搜尋盤面上另外一個沒標號的位置作擴散動作。

為了避免接近終局時，雙方棋子過度分散，導致搜尋很深及考慮威脅度都無法探測到對方子的存在如圖 3-32 所示，所以我們增加了盤面位置分數，這個策略主要是參考了東華大學賴學誠所撰的論文”電腦暗棋程式與經驗法則配合之實作”[5]，由於棋子在邊邊角角活動性較低，所以我們會給邊邊角角比較小的分數如表 3-2 所示，所以若如圖 3-32 所示，黑方得到了位置分數 2+3，紅方則是 2 。

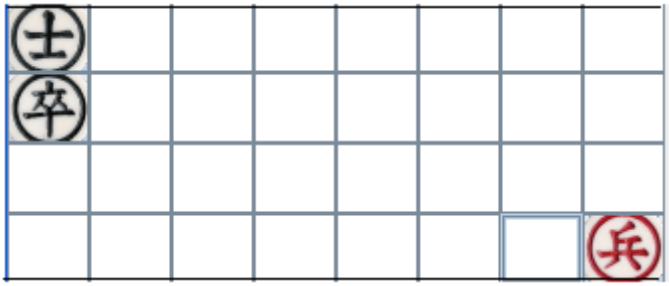


圖 3-32 棋盤位置範例

表 3-2 棋盤位置分數

2	3	3	3	3	3	3	2
3	4	4	5	5	4	4	3
3	4	4	5	5	4	4	3
2	3	3	3	3	3	3	2

由於位置分數給的很小，所以只有在吃子和威脅度都無法呈現出盤面好壞時才會有效果。此法會讓邊角的棋子在幾乎不影響盤面好壞的情況漸漸往活動性高的中間區域移動。

第九節　翻棋策略

一般人類玩家通常會考慮翻棋的情況都是無論明棋怎麼走時的利益都比翻棋差時才會翻棋，且當所翻之子所蒙受最大損失會低於明棋走步的情況下才會傾向翻棋，所以我們對每個暗棋賦予所有未翻開的棋種去做淺層搜尋，去計算每個暗棋位置翻棋所蒙受的最大損失，取最大損失為最低的點為翻棋點，若最大損失皆相同的情況下，則考量利益損害總和為最高的點為翻棋點，在當翻棋點的最大損失優於明棋搜尋時，代表翻這個點不可能會有比明棋走步更糟的結果，所以就

會在此點翻棋，反之則走步。

圖 3-33 為一翻棋範例，若現在由紅方翻棋，則最大損失最低的點共有 5 個，分別為 D2、F2、H2、B4、D4，其中紅方翻開 D2 的利益損害總和為紅炮吃到黑馬*2，因為紅炮可以吃到黑馬，而紅炮有兩個未翻，所以加總起來為兩顆黑馬的價值，而紅方翻開 F2、H2、D4 的利益損害總和以上述方法類推為紅炮吃到(黑車+黑馬)*2，皆大於紅方翻開 D2 的利益損害總和，所以我們程式翻開了 F2。

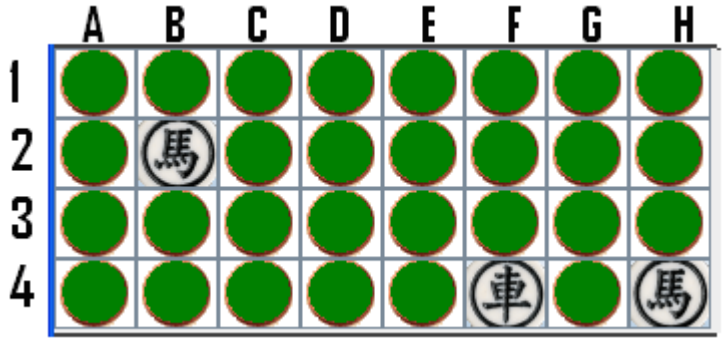


圖 3-33 翻棋位置範例 1

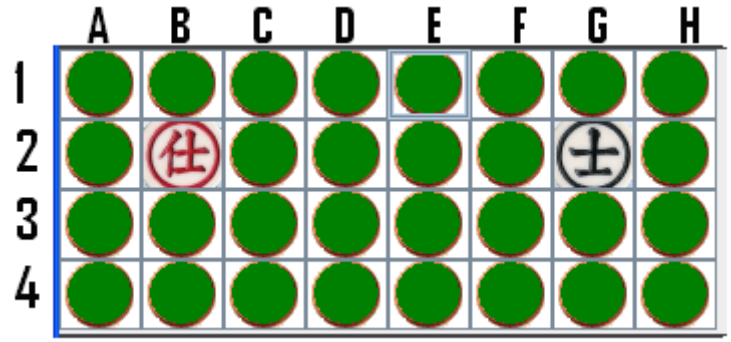


圖 3-34 翻棋位置範例 2

圖 3-34 為另一例子，若輪到黑方翻棋，雖然黑士周圍的位置可以吃子機率很高，但是最大損失卻有可能是一整顆黑士，所以我們不會考慮這一點，而 D2 翻開不會有任何風險且有可能翻開黑包吃到紅仕，由於黑包有兩顆未翻，所以利益總和為(黑包吃紅仕)*2，我們的程式選擇了沒有風險且利益損害總和為(黑包吃紅仕)*2 的 D2 來翻棋。

第四章　結論與未來方向

第一節 結論

在我們所收集關於電腦暗棋的相關文獻中，對子力分數都是採取固定的分數，所以我們希望能夠在盤面情況不同時，動態的去調整每顆棋子的分數，以期能夠更謹慎精確的審視盤面，以用來提高電腦暗棋棋力。

我們所撰寫的暗棋程式，在 CPU 為 2.80GHz、記憶體為 1.24GB 的環境下，初步與初學人類玩家(即本論文作者與其他同學)進行了平均10秒內出手的12盤快棋實戰測試，其測試結果如下：

	勝	敗	和
我們的程式 vs 人類玩家	6 盤	4 盤	2 盤
謝曜安的程式 vs 人類玩家	8 盤	8 盤	4 盤

雖然樣本數不及謝曜安的樣本數，不過經由我們的初步測試其與人類玩家對戰的戰績(若扣掉和棋)有六成左右略高於謝曜安的程式與人類玩家對戰的五成。

而與謝曜安程式的對戰中勝率也高於五成，其測試結果如下：

	勝	敗	和
我們的程式 vs 謝曜安的程式	7 盤	4 盤	4 盤

第二節 未來研究方向

目前我們對於動態子力的考量，主要只有針對吃子關係也就是每顆棋子的攻擊力去做考量，不過每顆棋子除了攻擊力之外也有防禦力的概念存在，例如帥就只能被卒和將吃，卒卻會被大部分的棋子吃掉，若能夠將防禦力和攻擊力的概念做個更好的整合，相信能夠產生更精確的動態子力之審局函數。

如果能夠將對局樹進行適當的切割交給不同處理器去做搜尋，將之平行化[12]，相信能夠在同樣時間下提高搜尋深度，並提高棋力。

還有如果能夠將重要路徑作延伸搜尋而不重要的路徑作較低深度的搜尋，相信這樣也能夠相當程度提高暗棋程式的棋力。

參考著作

[1] “Wikipedia”，網址：http://zh.wikipedia.org/wiki/。

[2] “象棋百科全書”，網址：http://www.elephantbase.net/index.htm。

[3] “Chess Programming wiki”，網址：http://chessprogramming.wikispaces.com/。

[4] 謝曜安，“電腦暗棋之設計及實作”，國立臺灣師範大學資訊工程研究所碩士論文，2008。

[5] 賴學誠，“電腦暗棋程式與經驗法則之配合與實作”，國立東華大學資訊工程研究所碩士論文，2008。

[6] 黃文樟，“電腦象棋深象中局程式的設計與實作”，國立臺灣師範大學資訊工程研究所碩士論文，2006。

[7] 王小春，“人機博奕”，重慶大學出版社，2002 年 6 月。

[8] 吳身潤，“人工智慧程式設計”，維科圖書，2002 年 3 月。

[9] Stuart Russell , Peter Norvig，Artificial Intelligence：A Modern Approach 2/E，PEARSON，2003 年 12 月。

[10] 何宏發、謝秋桂，“電腦象棋-原理、設計、實作及工具箱”，第三波出版社，1988 年 12 月。

[11] 方裕欽，“UCT 算法的適用性及改進策略研究－以黑白棋為例”，國立臺灣師範大學資訊工程研究所碩士論文，2008。

[12] 林子哲，“「深象」象棋軟體平行化之研究”，國立臺灣師範大學資訊工程研究所碩士論文，2007。

[13] 涂志堅，“電腦象棋的設計與實現”，中山大學碩士論文，2004。

[14] 郭哲宇，“電腦象棋擴大空步剪裁演算法的設計及實作”，國立臺灣師範大學資訊工程研究所碩士論文，2007。