第三章 搜尋演算法

搜尋演算法與審局函數是電腦象棋中局程式的兩大部份，決定了整體程式棋力的強弱，其中搜尋演算法可說是程式最主要的骨幹，一般而言，搜尋速度的快慢決定了程式在給定時間內所能搜尋的層數多寡，而棋力的高低則與搜尋層數成正比[8]。

本章從基礎的對局樹搜尋演算法 Min-Max 介紹起，並說明程式中真正使用的搜尋演算法 NegaScout。此外，程式中所使用的搜尋演算法及相關技術，主要都是在徐讚昇教授與許舜欽教授的指導下，由吳光哲學長所完成，特此提出，並表達感謝之意。

第一節 Min-Max 搜尋演算法

在棋局的進行過程中，輪流由紅黑雙方走子，並謀求對己方能獲得最大利益的走法，所以我們可以將對局的進行過程視爲一棵樹，以棋局開始盤面爲根節點，將所有可能走法展開，在雙方都選擇最佳走法的情況下，只要拜訪所有節點一次，就可以得知在開始盤面下最佳的走法。爲了區別所謂的最佳盤面，必須有特定的“審局函數”，能夠對每一個盤面給予適當的分數，程式中以零分爲分界點，正分代表紅方較好，負分則代表黑方較好，所以在棋局進行的過程中，紅方盡量要使盤面分數越大越好，而黑方則剛好相反，因此，在輪流走棋的過程中，如果輪到紅方走棋，那麼在該盤面下應選擇可以獲得最大分數的走法；反之若輪

到黑方走棋，則必須選擇該盤面下能夠獲得最小分數的走法，這也就是 Min-Max 搜尋演算法[6]的精神。

以圖 3-1 爲例，假設紅黑雙方輪流各下一手棋之後共有四種可能盤面，而每個盤面根據審局函數給予不同的評分，最後根據每個節點輪紅方或黑方走棋，分別選擇能夠得到最大或最小分數的走法，如此往上遞迴選擇，便能得到根節點處對紅方而言最好的走法。

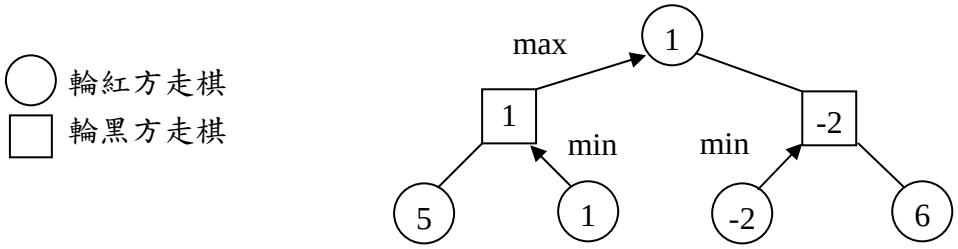

圖 3-1 Min-Max 搜尋演算法示意圖

第二節　Alpha-Beta 搜尋演算法

Min-Max 搜尋演算法的精神雖然正確，但是因爲每一個節點都必須拜訪過一次，所以耗費的時間相當多，實際上在搜尋過程當中，並不需要每個節點都拜訪過一次，也不會影響原來的結果。Brudno[1]在 1963 年首先發表了第一篇 Alpha-Beta 演算法的論文，1975 年 Knuth 和 Moore[7]在數學上證明了 Alpha-Beta 演算法的正確性。

Alpha-Beta 演算法是通過下邊界 Alpha 及上邊界 Beta 對最後搜尋所得到的結果，限制其範圍，由於 Alpha-Beta 是深度優先搜尋，如果在搜尋過程中發現某些後來搜尋的子樹結果不會落在上、下邊界之內，則該子樹不會影響最後搜尋結

果，就不必搜尋該子樹(cutoff)，節省搜尋的時間。

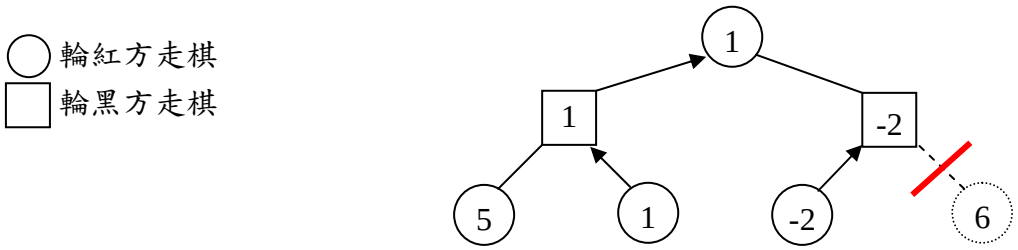

圖 3-2 Alpha-Beta 搜尋演算法示意圖

Alpha-Beta 演算法搜尋的效率與搜尋走法的順序有很大的關係，如圖 3-2 所示，如果右半邊子樹盤面分數-2 與 6 的節點互換順序，就不會發生裁剪(cutoff)，和原來 Min-Max 搜尋演算法搜尋的節點數目就會相同。在最佳走法搜尋順序時，Knuth 和 Moore 證明了 Alpha-Beta 演算法最少必須搜尋的節點數目爲：

$$\begin{cases} n = 2b^{d/2} - 1 & \text{(d 為偶數)} \\ n = b^{(d+1)/2} + b^{(d-1)/2} - 1 & \text{(d 為奇數)} \end{cases}$$

其中 n 代表搜尋節點數目，b 代表 branch factor， d 代表搜尋深度。

第三節　NegaScout 搜尋演算法

在 Alpha-Beta 演算法中，上邊界 Beta 及下邊界 Alpha 的大小，直接影響到發生裁剪的機率， Alpha 及 Beta 之間的距離，稱爲“window”，如果 window 的值越小，發生裁剪的機率就越高。除了 window 的大小之外，走法排列的順序也會影響到發生裁剪的機率，如果排列得好，可以裁剪掉絕大多數的子樹，也就能提高搜尋的效率。NegaScout 演算法[7]就是基於以上這兩個假設，對 Alpha-Beta 演算法進行改良。

NegaScout 演算法假設最佳的走法都排在前面，所以當搜尋過程遞迴拜訪子樹時，第一個子節點採用正常的 Alpha-Beta window 展開，而其餘的子節點則以[Beta-1, Beta] window 展開，由於不可能有某個整數大於 Beta-1，但又小於 Beta，所以用[Beta-1, Beta] window 展開的子樹不會得到真正的搜尋分數，[Beta-1, Beta]所形成的 window 也稱爲“minimal window”。以 minimal window 展開的子樹雖然不會得到真正的搜尋分數，但因爲假設最佳走法都排列在前面，所以後面的子樹只要證實搜尋結果大於等於 Beta，產生裁剪即可，此種裁剪稱爲 Beta cutoff(搜尋得到的分數大於等於 Beta)。如果用 minimal window 搜尋之後發現不會產生 Beta cutoff，那麼代表無法裁剪，必須重新用正常的[Alpha, Beta] window 重新搜尋這棵子樹，以得到確切的搜尋分數。

雖然 NegaScout 演算法可能會重複搜尋同一棵相同的子樹，但只要走法排序夠好，裁剪的機率很高，相對重複搜尋的機率就降低了，搜尋速度比 Alpha-Beta 演算法快得多。

表 3-1 爲 Alpha-Beta 演算法與 NegaScout 演算法的平均搜尋節點數比較。

表 3-1 NegaScout 與 Alpha-Beta 演算法平均搜尋節點數比較表

搜尋深度	平均搜尋節點數		減少百分比
	Alpha-Beta	NegaScout	
5	56495	38296	32.2%
6	578853	214901	62.8%
7	934532	371265	60.2%
8	3618750	1244298	65.6%
9	18392301	5928621	67.7%
10	34599490	12591827	63.6%

第四節　Transposition Table

在搜尋的過程中，由不同的走法可能會走到相同的盤面，如圖 3-3 所示。

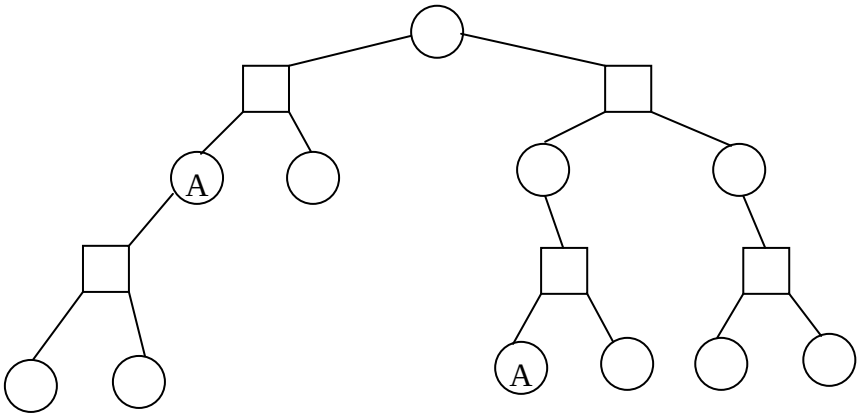

圖 3-3 搜尋中重複盤面示意圖

換句話說，在搜尋的過程當中，往往會浪費時間重複搜尋相同的子樹。爲了避免發生這種情況，我們將搜尋過的盤面儲存在一個 hash table 中，每次展開所有走法之前，先試著從 hash table 中查詢是否已經搜尋過這個盤面，如果是的話，就不必搜尋此盤面以下的整棵子樹；反之，若是在 hash table 中找不到此盤面，便將此盤面底下的整棵子樹展開，並將最後搜尋得到的值存在 hash table 中，此 hash table 便稱爲“同形表”(transposition table)[3, 14]。使用同形表可大幅減少搜尋的節點數，表 3-2 是使用與不使用同形表平均搜尋節點數的比較。

表 3-2 使用同形表平均搜尋節點數比較表

搜尋層數	平均搜尋節點數		減少百分比
	不使用同形表	使用同形表	
7	2972311	820305	73.17%
8	9018695	2355633	74.81%
9	17031273	4492392	74.65%
10	64989394	10635712	84.19%

除了在 NegaScout 搜尋中使用同形表之外，在產生所有走法時，也可以利用同形表減少搜尋節點數。在展開所有走法時，先往前展開一層，對這些走法能到達的盤面，在同形表中查詢，如果能在同形表中找到且會引發裁剪，那麼這些走法就不必繼續往下展開，直接回傳同形表中儲存的分數即可，如此一來，若此盤面共有 m 種走法，而在第 k 個走法發現可利用同形表中儲存的值引發裁剪，那麼可省下原先前(k-1)種走法搜尋子樹的時間，此作法稱爲 enhanced transposition cutoff [11]，圖 3-4 爲示意圖，表 3-3 則是使用與不使用 enhanced transposition cutoff 的搜尋效率比較。

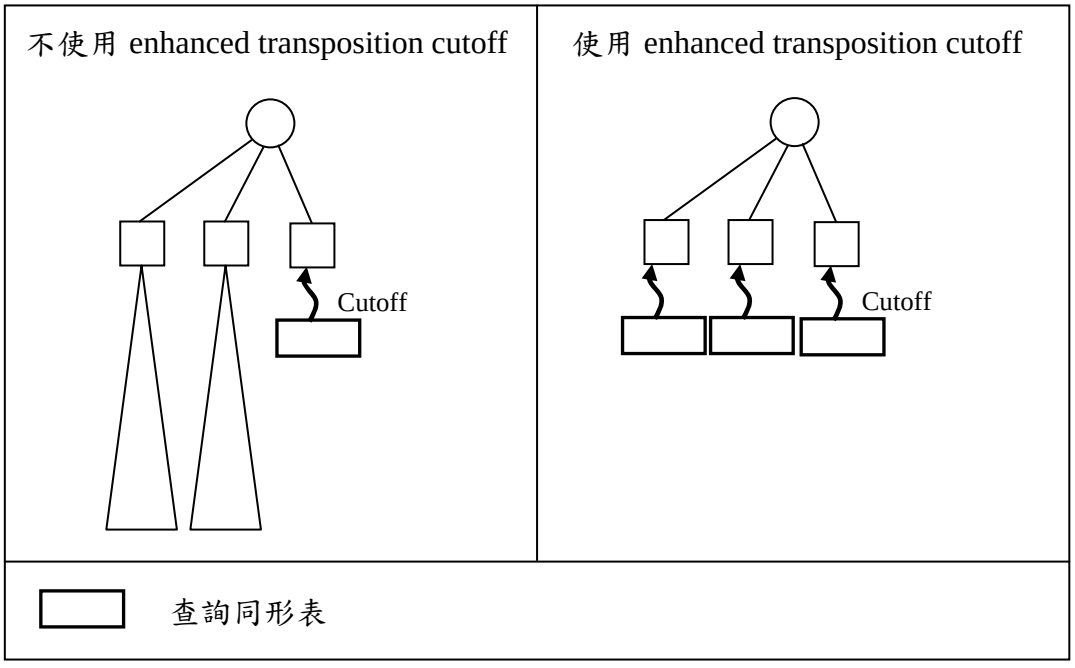

圖 3-4 enhanced transposition cutoff 示意圖

表 3-3 enhanced transposition cutoff 平均搜尋節點數比較表

搜尋層數	平均搜尋節點數		減少百分比
	不使用 enhanced transposition cutoff	使用 enhanced transposition cutoff	
5	94943	92750	2.3%
6	312463	299810	4.04%
7	820305	797297	2.8%
8	2355633	2271273	3.58%
9	4492392	4316250	3.92%
10	10635712	10273562	3.4%

第五節　其他搜尋技術

一、逐層加深(iterative deepening)搜尋[14]：

在實際應用上，對特定的盤面進行搜尋，通常是有時間限制的，爲了達到此目的，必須給定有限的搜尋層數 d，但由於不確定搜尋完 d 層必須花費多少時間，所以利用逐層加深搜尋，一開始先搜尋一層，如果搜尋完一層時間還沒用完，就繼續搜尋第二層，依此類推，一直到所分配的時間用完爲止，如此一來，自然可以達到時間控制的目的。逐層加深搜尋的另一個好處，則是可以配合同形表的使用，在搜尋第 k 層時，優先搜尋第(k-1)層的最佳走法，通常這樣可以提高裁剪的效率，減少搜尋的節點數目。

二、寧靜搜尋(quiescence search)：

在時間限制之內，只能搜尋有限的層數，如果在最後一層發生兌子的情況，因爲時間已經用完，搜尋層數不足以看到下一手對方吃回我方子力，會誤以爲己方佔優，這種情況，一般稱之爲水平效應(horizontal effect)，因爲搜尋層數的不足，

而對盤面形勢造成誤判。爲了解決此一問題，有人提出了寧靜搜尋(quiescence search)[5,10]，寧靜搜尋可以視爲一種選擇性搜尋，只針對吃子步及解將步作展開，最後到達寧靜盤面(沒有子力交換的盤面)爲止。正常 NegaScout 搜尋層數搜尋完時，若此時爲非寧靜盤面，則接著由寧靜搜尋進行搜尋，如圖 3-5 所示。

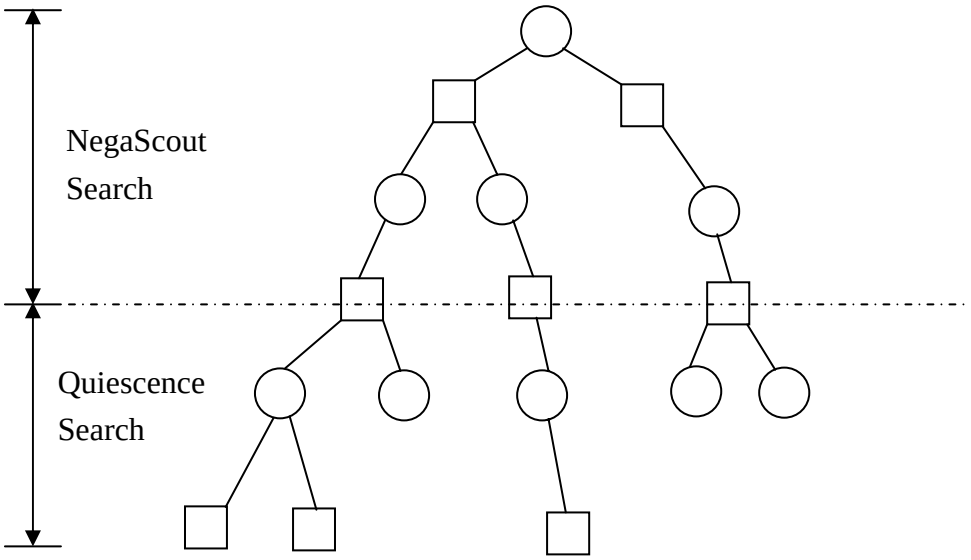

圖 3-5 寧靜搜尋示意圖

三、延伸搜尋(extension search)：如同圖 3-5 所示，真正搜尋過程中的遊戲樹其實是參差不齊的，在某些情況下，可以針對必要的部份加深搜尋深度，提高盤面評價的準確性，例如上述的寧靜搜尋是其中一個例子。但也不能對太多節點作延伸搜尋，否則會因爲搜尋節點數目成長過快，導致搜尋效率低落。目前程式中使用兩種延伸搜尋技術，第一種是將軍延伸，如果遇到將軍盤面，因此時必須解將，而通常解將步的步數並不多，所以此時加深搜尋深度一層，並不會增加太多搜尋的節點數，但又能對將軍攻擊的威脅性作更準確的評價。第二種則是單步延伸，當某方被將軍，且解將步只有一步時，除了將軍延伸之外，額外再加深一層

的搜尋深度，稱爲單步延伸。

四、空著裁剪(null move pruning)：

Null Move 最早在 1989 年被 Beal 提出[4][5][9]，Null Move 主要精神是當己方不走子，由對手連續走兩步時，如果減少搜尋深度的淺層搜尋情況下，搜尋分數也會超過 Beta，那麼可以對此走法進行裁剪，因爲這表示對手即使連走兩步，情勢也不會對對手比較有利，那麼此走法很可能是一步不好的走法。使用 Null Move 進行裁剪的好處是，可以裁剪掉大量不好的走法，減少搜尋的節點數，從而提高搜尋效率，但要注意的是，在某些情況下不能使用 Null Move 進行裁剪，例如在 Zugzwang 盤面中便不應使用，所謂的 Zugzwang 所指的是被迫走子，即每一步都對己方不利，但卻又不得不走。在中國象棋中，常常在殘局時遇到即使連走兩步也不一定能夠佔優，甚至可能有輪到誰走誰不利的局面，此時就不應該使用 Null Move。