第二章　相關文獻及基礎理論

第一節　電腦黑白棋發展回顧

以下我們介紹一些具有代表性的電腦黑白棋程式及相關研究成果：

1. Logistello

Logistello 可以說是黑白棋程式的一個里程碑，它引進了全新的模式(pattern)評估法[7]、快速的 MPC（Multi-ProbCut）搜索演算法[8]、自我學習功能[9]等多項技術，不僅使速度大大提高，還使黑白棋程式的棋力有相當的飛躍。1997 年 8 月，Logistello 以 6:0 擊敗世界冠軍 Takeshi Murakami，從此黑白棋程式便把人類棋手遠遠甩在後面。

Logistello 是最強的黑白棋程式之一，在 1993 年到 1997 年的 25 場比賽中，Logistello 有 18 場獲得冠軍、6 場獲得亞軍。在功成名就之後，Logistello 於 1998 年 1 月宣佈退休，把發展空間讓給其它的黑白棋程式。

2. WZebra

大名鼎鼎的黑白棋程式「斑馬」，WZebra 的核心程式是 Zebra，WZebra 是其 Windows 版本。WZebra 基本沿用了 Logistello 的設計理念和技術，而在功能和介面上更強大。除了可以下棋外，Wzebra 還提供了打譜、棋局分析、自我學習等功能，甚至可以載入 Thor 棋譜檔，進行針對性訓練，是目前功能最全面的黑白棋程式。

WZebra 的棋力也非常強勁，在標準比賽時間（雙方各有 15 分鐘時間、超時作負）內，WZebra 的搜索深度可以達到中局 18 到 27 步、終局 24 到 31 步。在 2003 年 4 月的 GGS 公開賽中，名列第三。更吸引人的是，WZebra 是自由軟體。WZebra 還提供了中文功能表，並提供中文說明，非常適合國內黑白棋愛好者使用。

3. 相關研究成果

黑白棋研究在 1997 年 Logistello 擊敗人類世界冠軍達到高峰，其作者 Michael Buro 更提出多項技術，其研究成果如模式(pattern)評估法[7]、快速的 MPC（Multi-ProbCut）搜索演算法[8]、自我學習功能[9]等，都深深地影響後來的電腦黑白棋發展。然而在 Michael Buro 之後，相關黑白棋研究文獻相當的少，大多都以電腦黑白棋程式設計為主。

第二節　電腦對局理論基礎

1. 位棋盤

位棋盤(bit board)常用在棋類的遊戲，如西洋棋、西洋跳棋及象棋，是由一串的位元表示棋盤的位置或狀態，在大量運算時，可以加速運算速度及減少記憶體、硬碟的使用量，目前已被使用在許多著名的棋類程式上。位棋盤如此的受歡迎，是因為只需透過僅一次的邏輯的運算，就可以取得我們想知道的盤面的狀態。以西洋棋來說，西洋棋程式想要知道盤面中間區域是否有兵的棋子，我們只需進行一次 AND 邏輯運算比較這區域的位棋盤狀態就可以得知。而在黑白棋上使用位棋盤，同樣也可加速運算速度、減少硬碟及記憶體使用量。

2. 對局樹

何謂對局樹（game tree）？假設輪到我方著手，我們會考慮走哪一步會比較有利，而我們走了這一步之後，對方有幾種走法，當對方走了某一步之後我方又該如何對應等，如將這一連串的對弈過程以樹狀架構表現出來，可稱此樹狀架構為對局樹。對局樹是十分重要的，在棋類遊戲中一般是不可能將所有的變化都搜尋一遍的，假設一步棋有 10 種走法，而雙方各要走 10 步棋才能分出勝負的話，那麼就一共有 10^{20} 種狀態需要搜尋。這樣的天文數字根本就不可能完成。於是在競局的搜尋法中，如何修剪所要搜尋的對局樹，來減少搜尋的量，成為最重要的問題。一般的方法，就是只考慮下幾步棋，再加上大略的棋盤審局函數，使局面保持優勢來繼續。由於人類棋手在下棋時大多也是採取這種思考模式，所以搜尋的深度和棋盤審局函數的好壞，便直接影響到程式在與人類對奕時的棋力。

3. 極大極小搜尋演算法

依照搜尋的目的不同，發展出不同的搜尋方法。常見的搜尋目的有 3 種：尋找路徑、尋找最佳路徑、和競局。由競局所發展出來的搜尋方法很多，包括了極大極小搜尋法（mini-max Search）[2]、α-β 修剪搜尋法（alpha-beta pruning）[4]、negascout 搜尋法和經驗式修剪搜尋法（heuristic pruning）[10]等等。這一類的搜尋法在棋戲中相當普遍，尤其是西洋跳棋和西洋棋。

而下棋時是一人走一步的，雙方都想要使自己的局面保持優勢，同時使對方處於劣勢。於是我們往往要考慮對手的應對走法，來決定下一步要怎麼走。同樣

地，對手也會預測我們的走法，不讓你稱心如意。極大極小法就是假設敵我雙方都選擇最佳走法的一種有限深度、深度優先的搜尋方法。從根節點出發，呼叫走法產生函式產生目前盤面可能走法，往下展開到某深度後，在葉節點處使用審局函數判斷局勢好壞。我方走法的選擇以得到最大分數走法為優先，反之對手方的目標則是選取使審局函數得到最小值的走法。因此程式在不同層中，交互使用取極大值和極小值的走法。這也就是這個名稱的由來。

如圖 2-1，只搜尋一層的時候，應該選取對自己最有利的走法，也就是中間的節點，可以得到 9 分。

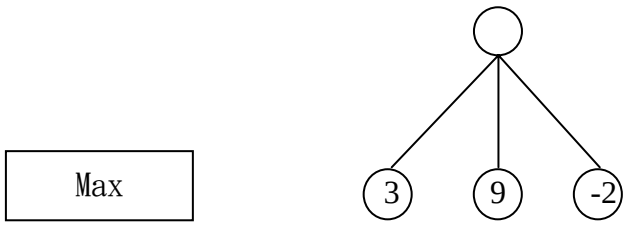

圖 2-1 搜尋 1 層

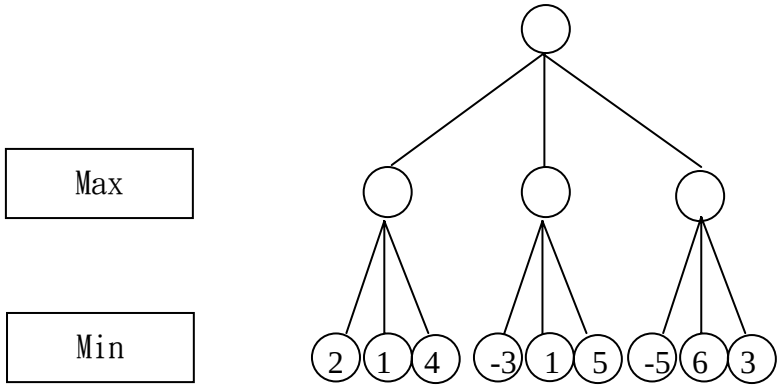

圖 2-2 搜尋 2 層

但是如圖 2-2 在搜尋兩層的時候，發現原先選擇的中間節點竟使我方反而只得到-3 分。因此我們應該選擇最左邊的節點，即使在最差的情況下，還是可以得到 1 分。

使用極大極小搜尋法[2]，在某一個深度內的所有變化都要一個一個地去檢查。然而真的有這個必要嗎？已知目前盤面有許多不同的走法，如果我們發現目前我們所看的走法中，使得剩下的走法無論如何都不能更好時，我們便可利用分支限制（branch-and-bound）的技巧，來增加極大極小搜尋的效率。也就是說，在搜尋的過程中，一旦發現目前所看的這個走法，已經比先前已知的走法還差的時候，就可以提早放棄它，進行切捨(cutoff)。這是因為敵我雙方輪流下子，雙方的立場不同，我方盡力使審局值拉高，而對方則是拼命使審局值降低。當發現無論如何都不能改變目前的最佳值時，當然也就可以提早放棄。

4. 同形表

在電腦棋類程式中，Hash Table 的使用通常被用來加快遊戲樹展開的速度，由於遊戲樹的展開通常十分龐大，不同的走步順序可能會導致產生相同的盤面狀況，如圖 2-3，倘若沒有一個表格記錄這些已展開子樹的資訊，當對局樹展開到之前已經處理過的盤面時，程式在未知的狀況下仍會繼續的展開，便會造成時間的浪費。為避免類似這種情況的發生，Hash Table 的使用是極為重要的。

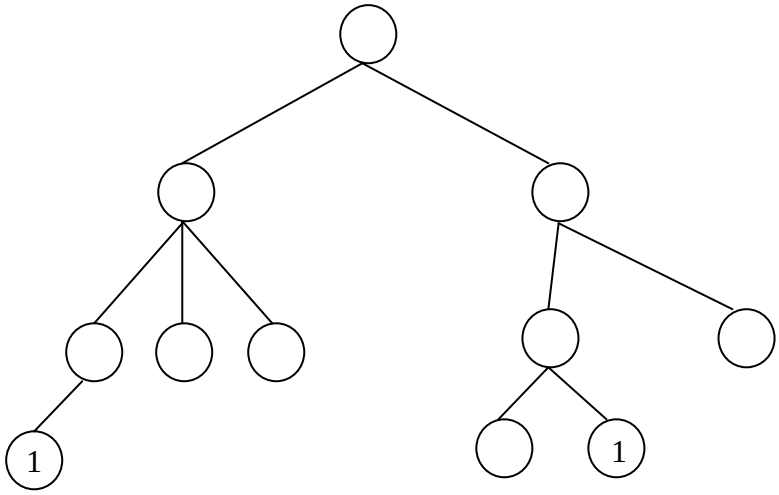

圖 2-3 重複盤面

每次展開盤面時，會先試著從 Hash Table 中尋找是否先前已展開過相同盤面，若已展開過，則直接從 Hash Table 中取出先前的結果，不用再重新搜尋一次；若找不到，則繼續展開目前盤面，並且把最後結果儲存在 Hash Table 中，供以後有相同盤面時使用。我們稱此 Hash Table 為同形表(Transposition Table)[11]。

而在目前眾多的電腦棋類程式，幾乎是每一套程式都會實作 Hash Table。然而，Hash Table 的實作過程中，很重要的一點是如何運用較好的方式來進行 hashing，所謂的較好是指每筆資料對應的 hash key 碰撞機會較少，使得資料能平

均分散。目前有許多的 hashing 方式運用在棋類程式上，較為大家採用的是 Zobrist hashing[1]方式。

5. UCT 演算法

UCT(Upper Confidence bounds applied to Trees)演算法是一種蒙地卡羅方法的擴展應用，是由兩位匈牙利研究人員 Levente Kocsis 與 Csaba Szepesvari 發明的一種新的演算法[3]。後來由 Sylvain Gelly 與 Yizao Wang 實作在圍棋程式 MoGo 中[15]，而 2007 年的春季，MoGo 在 9 路圍棋的比賽中擊敗了實力強勁的業餘棋手，在 19 路圍棋的比賽中也擊敗了實力稍弱的業餘棋手，充分展示 UCT 演算法的威力。而 UCT 演算法的主要精神是對當前候選的走步進行大量的模擬，而這樣的模擬必須達到一種平衡，既要嘗試目前的最佳走步，還要搜索目前非最佳的走步，以發現其中可能隱藏的好步，以確保不會因為先前的估計錯誤而錯失妙招。

UCT 演算法會為每一種走步計算出一個索引值，然後按照索引值最高的走步出招。索引值的決定則是由該走步的贏率與修補分數二部分共同加總決定。然而為何不單單考慮走步的贏率而有修補分數的出現呢？這是為了避免 UCT 演算法會一直選擇贏率高的走步進行模擬，不會選擇贏率低的走步進行模擬，因而錯失當前贏率低走步的好步。所以 UCT 演算法設計了一個機制，在選擇走步時，使得計算次數少的候選走步的修補分數較大，以提升選擇到拜訪次數少的走步的機率，透過這樣的機制，會使贏率的總體分布趨向平坦。

在大量模擬的過程中，UCT 演算法會建立一棵決策樹，用來記錄每一種走步

的拜訪次數、得到的分數與其他資訊。決策樹建立的過程中，會選擇當前走步的最佳索引值，當選擇到一種先前從未計算過的走步時，UCT 演算法就會將它納入決策樹中，並利用蒙地卡羅法以隨機的方式求得該走步最後的勝負狀況。接著，UCT 演算法就會由下而上地更新資訊，從根節點走至未計算過走步路徑上的所有走步均會被更新，以提供下次模擬時使用。

以下我們舉一個範例說明決策樹建構的過程，如圖 2-4。其中節點內的數字如 1/1、0/1 等，其意義為該節點獲得分數/該節點被拜訪次數。一開始對某盤面進行 UCT 模擬，因此只有一個根節點，如圖 2-4(a)。接著在決策樹中納入了新的走步，如圖 2-4(b)，並對其進行蒙地卡羅法，最後將分數由該走步往上更新至根節點，如圖 2-4(c)。同理，再進行一次 UCT 模擬，在決策樹中納入了走步，並進行蒙地卡羅法，如圖 2-4(d)，得到的勝負結果，往上更新分數，如圖 2-4(e)。之後的步驟其實都相似，因此我們省略了納入決策樹與蒙地卡羅法的過程，直接將更新分數的結果秀出。一直進行到圖 2-4(g)為止，共模擬了 4 次，建立根節點下的 4 個子節點。

為了簡化說明，接下來我們在索引值計算部分，省略修補分數，只考慮贏率。如圖 2-4(g)，此時索引值最佳者為 1/1，即節點 A、B，因此任選一個節點進行。在圖 2-4(h)中，索引值最佳者為節點 B 的 1/1，因此會選擇節點 B。在圖 2-4(i)中，當選擇了節點 B 後，發現節點 B 的另一個分支節點 D 還沒被產生，因此將節點 D 納入決策樹中。在圖 2-4(j)中，根據索引值選擇了節點 B、C，並將節點 E 納入決

策樹中。

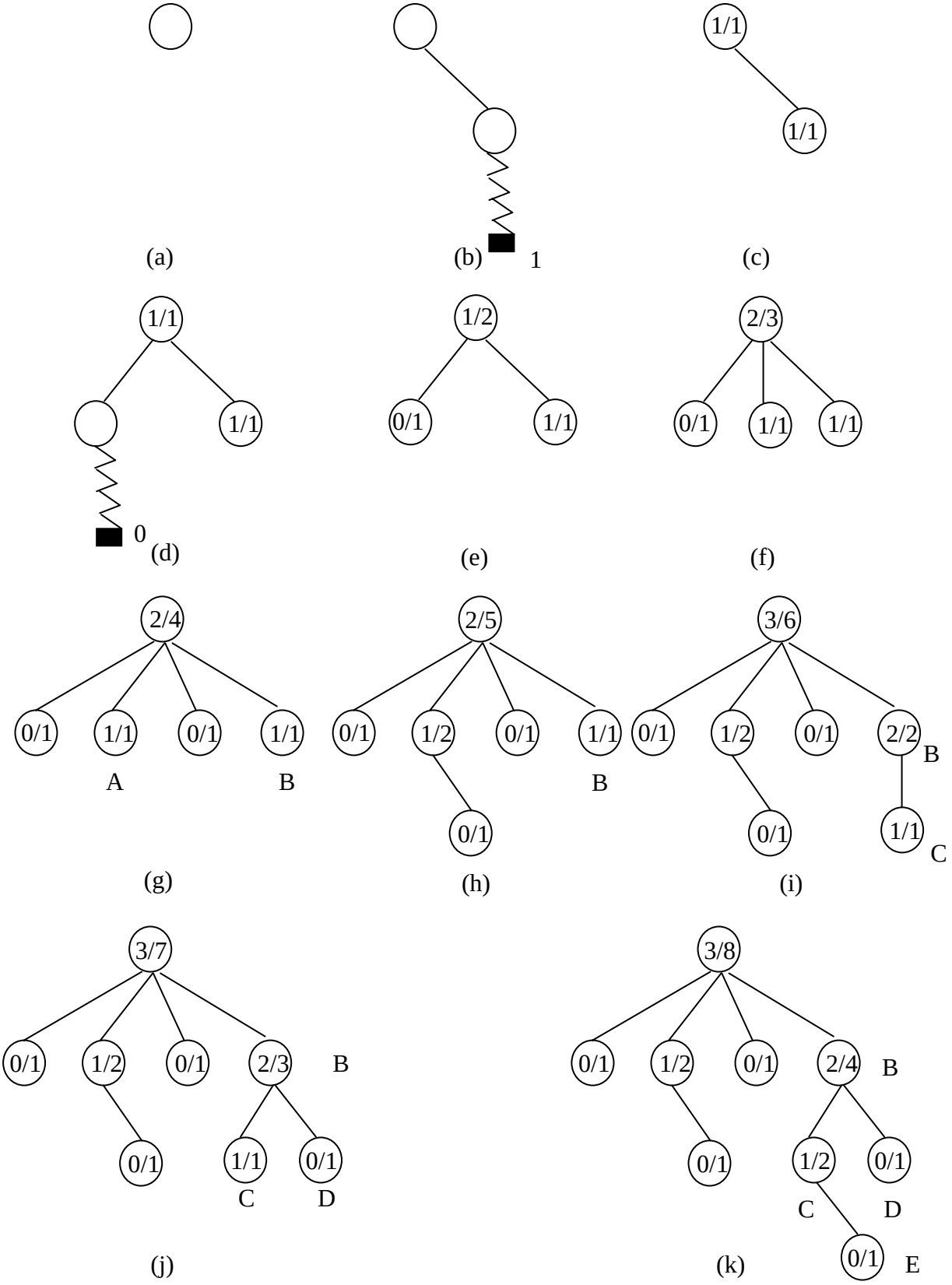

圖 2-4 UCT 演算法決策樹建立過程