第二章　搜尋演算法

第一節 簡介

棋類遊戲大都可以利用遊戲樹 (Game Tree) 展開節點，一個節點就代表一個盤面，我們利用井字遊戲(Tic-tac-toe)來做示範，如圖 1。

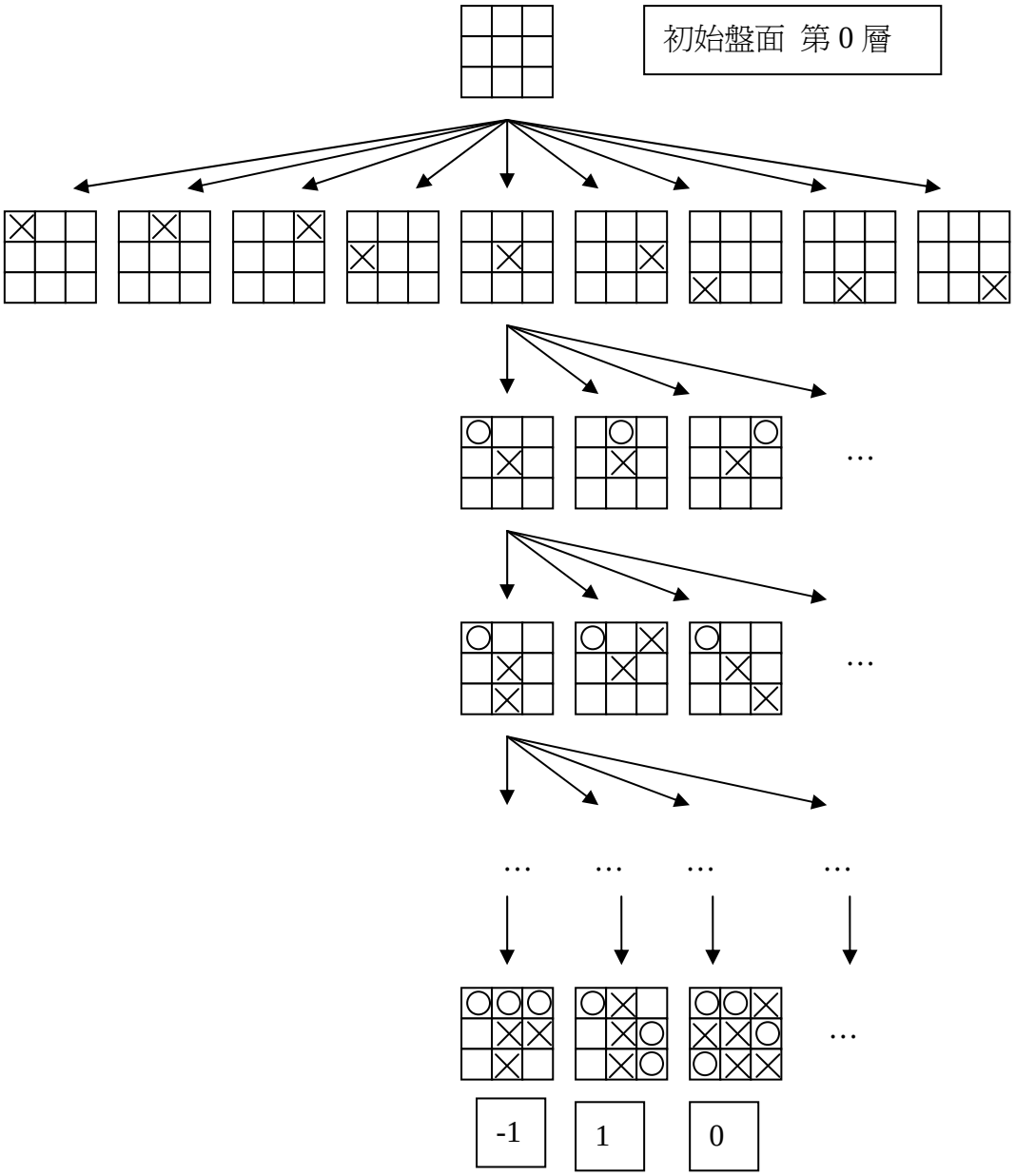

圖 1 Tic-tac-toe 遊戲樹

圖 1 是 Tic-tac-toe 部分的遊戲樹，奇數層為先手甲方可以走的盤面，偶數層為後手乙方可以走的盤面。最後的葉節點就利用審局函數給予盤面分數，甲方贏的話可以得到 1 分，甲方輸的話就得到-1 分，雙方平手就得 0 分，由此便可判斷誰勝誰負。我們可利用遊戲樹展開的方式，來搜尋全部的走法，尋找最佳的走步。

第二節 審局函數

Tic-tac-toe這個遊戲因為搜尋的深度最多只要 9 層，到達葉節點就可以知道誰勝誰負，但是，象棋遊戲搜尋的深度不止 9 層，而是數十層，甚至有時候會達到一百多層，以目前電腦的能力還不能把象棋遊戲樹完整的展開，就只能先展開至某一層(N層)，利用審局函數(Evaluation Function)算出搜尋至第N層節點的分數，並將分數回傳，再經由搜尋演算法比較之後，選出最佳的走法。如圖 2，遊戲樹展開，先由根節點A展開兩個分支B、C，接著B、C分別展開D、E、F和G、H五個分支的時候，假設達到最深深度，就呼叫審局函數，將盤面分數算出來，此例中D、E、F、G、H的分數分別為-5、19、-8、4、9。

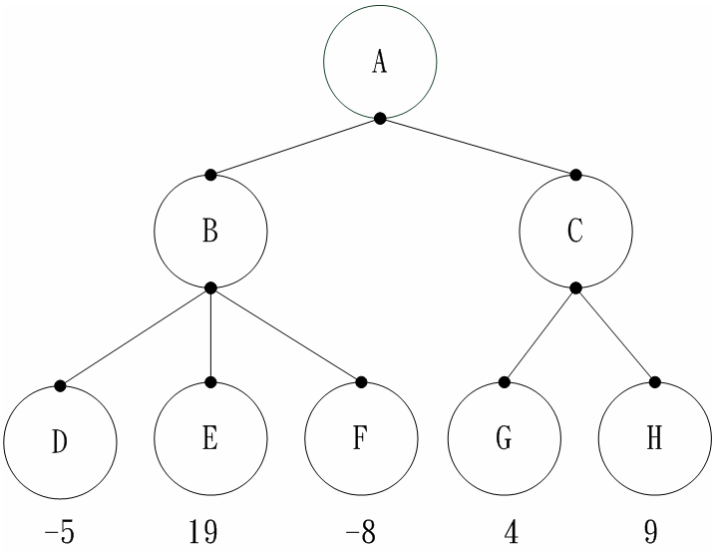

圖 2 審局函數

第三節 Mini-Max 演算法

當甲、乙兩人在下棋的時候，甲會想下出他覺得最好的走法，讓自己的局面變好，而乙也是想讓自己的局面變好，也就是想讓甲的局面變差。利用審局分數的想法，甲是想盡辦法將自己的分數變高，乙則是要讓甲的分數變低。

因此我們可以將遊戲樹的節點分成兩類：一種是取最大值與另一種是取最小值。圖 2 的遊戲樹就變成圖 3。方形節點是取其子樹的最大值，而圓形節點則是取其子樹的最小值。當搜尋深度到達預期的目標，則呼叫審局函數 Evaluate(node)，回傳盤面分數，D、E、F回傳其分數-5、19、-8 給B，B圓形節點取其子樹(D、E、F)的最小值-8，G、H回傳其分數 4、9 給C，C圓形節點取其子樹(G、H)最小值 4，B、C分別回傳其分數-8、4 給A，A方形節點取其子樹(B、C)的最大值 4，最後選擇的路徑為A→C。其虛擬碼如圖 4。

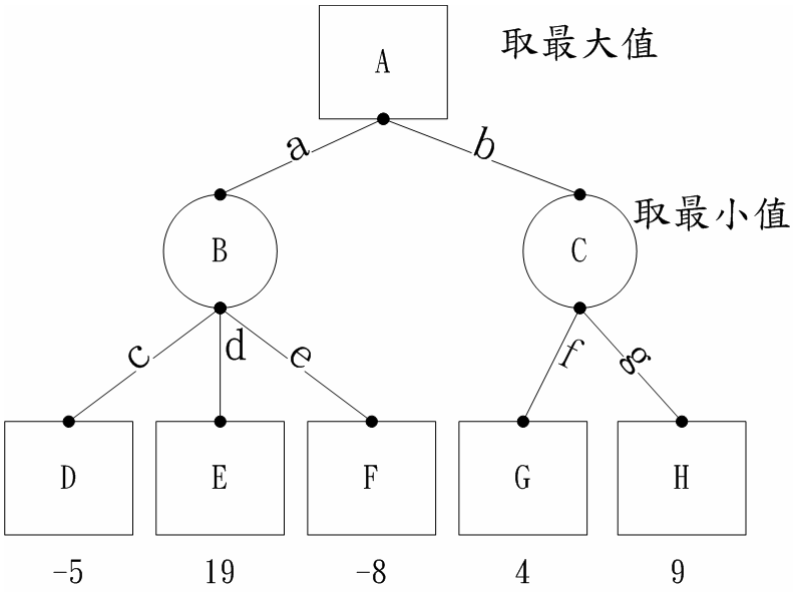

圖 3 Min_Max 遊戲樹

```
Mini_Max(node)
    v=Max_Value(node)
    return v

Max_Value(node)
    if(depth = =0)
        v=Evaluate(node)
    else
        v=-∞
    for x =1 to all children of node
        v = max(v , Min-Value(x))
    return v

Min_Value(node)
    if(depth = = 0)
        v=Evaluate(node)
    else
        v =  ∞
    for x =1 to all children of node
        v = min(v , Max-Value(x))
    return v
```

圖 4 Min_Max 演算法

第四節 Nega-Max 演算法

Mini-Max演算法可以簡化成不用判斷是取最大值或是取最小值，只要簡單的將回傳的值變號，並且取最大值，但是，在葉節點的時候必須注意，要以葉節點的觀點回傳分數，此演算法就叫做Nega-Max，演算法如圖 5。範例如圖 6，當c、d、e路徑展開之後，分數各別為-5、19、-8，回傳到其父節點的時候須要將其值變號，所以中間圓形節點的分數就是 8，當f、g路徑展開之後，分數各別為 4、9，回傳到中間圓形節點分數就是-4，當中間兩個圓形節點再回傳到根節點的時候再變號，所以根節點的分數就是 4 分，

```
Nega_Max (depth) {
   if ( depth = = 0 )
      if(turn = = RED)
            return Evaluate();
      else
            return –Evaluate()
   GenerateLegalMoves();                    //產生走子
   while ( move=MoreMoves() ) {             //選一個走步
      MakeMove(move);                       //走步
      v = max( v , - Nega_Max( depth - 1 );
      UnmakeMove();                         //還原盤面
   }
   return v;
}
```

圖 5 Nega_Max 演算法

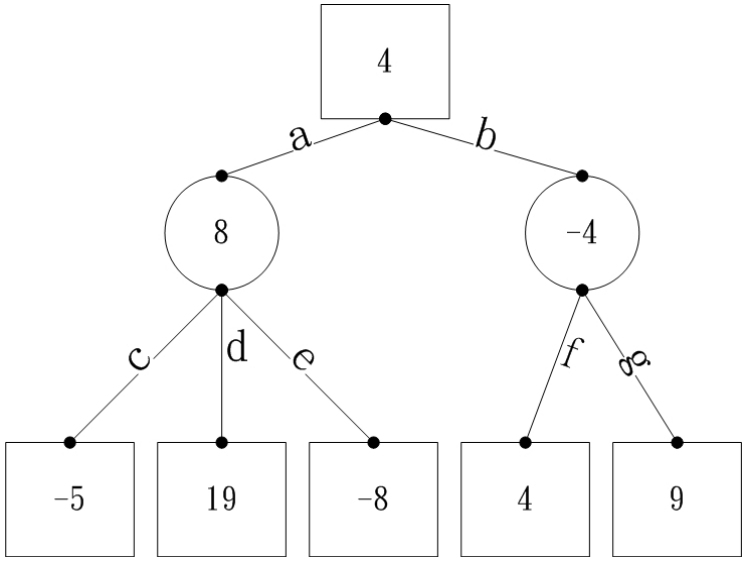

圖 6 Nega_Max 遊戲樹

第五節 Alpha-Beta 演算法

Min-Max和Nega_Max搜尋演算法可以找出如何走是最好的，但是它必須搜尋完整的遊戲樹，象棋遊戲樹中每個節點平均有 30 個走法，如果要搜尋N層，就要拜訪大約 30^N個節點，需要耗費相當多的時間。為了改進效率，Brudno發表了Alpha-Beta的演算法[5]，如圖 7。Alpha為搜尋分數的下界，Beta為搜尋分數的上界，當搜尋的分數不在Alpha、Beta分數之間，(當分數小於Alpha，這個分數太低對我方不好，當分數大於Beta，分數太高，對方不會輕易的讓我們獲得優勢)，就不必搜尋那些節點的子樹，以節省搜尋的時間。如圖 8，當c、d、e路徑展開之後得到分數 5、19、8，會將分數 5 傳回到根節點，此時根節點分數Alpha值≧5，當展開f路徑的時候發現分數 4 比根節點的Alpha值小，之後的g路徑就不用展開了。

```
Alpha_Beta (depth, alpha, beta) {
   if ( depth = = 0 )
      return Evaluate()
   GenerateLegalMoves();                    //產生走子
   while ( move = MoreMoves() ) {           //選一個走子
      MakeMove(move);                       //走步
      v = - Alpha_Beta( depth - 1, -beta, -alpha );
      UnmakeMove();                         //還原盤面
      if (v >= beta) {
         return beta;
      }
      if (v > alpha) {
         alpha = v;
      }
   }
   return alpha;
}
```

圖 7 Alpha_Beta 演算法

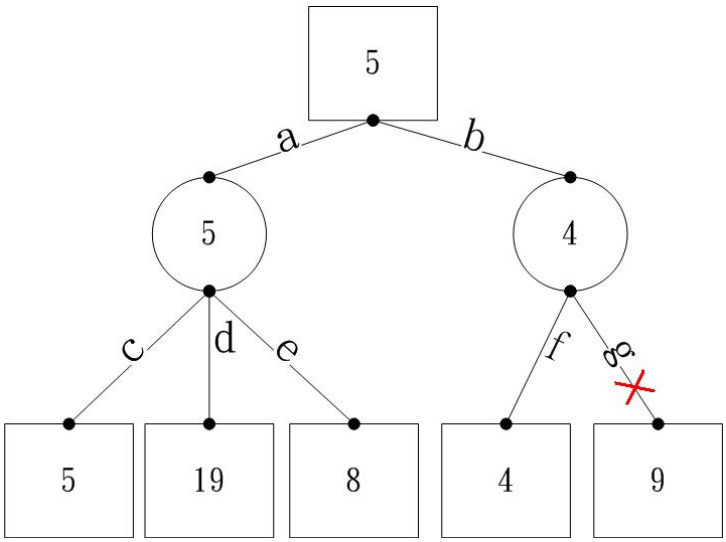

圖 8 Alpha_Beta 遊戲樹

第六節 Nega-Scout 演算法

Alpha-Beta 演算法，是依據 Alpha 和 Beta 的值來判斷是否要裁剪子樹。Alpha 和 Beta 的區間，我們稱為”window”。如果 window 越小，就越有機會發生裁剪(pruning)。當走法排序為最佳的時候，就可以裁剪最多的子樹。以減少搜尋的時間。

Nega-Scout 演算法[23]，見圖 9。每個子樹只有最左邊的主要變異(Principal Variation)路徑才會用完全的 Alpha-Beta window 展開，其餘的都是利用[-alpha-1, -alpha] window 展開，[-alpha-1, -alpha]這個 window 我們也把它稱為 null-window。因為我們假設最好的走法都排列在前面，當回傳的值大於 Alpha 且小於 Beta，才要另外再使用正常的 alpha-beta window 來做搜尋。

```
Nega_Scout (depth, alpha, beta) {
   if ( depth = = 0 )
      return Evaluate()
   GenerateLegalMoves();                    //產生走子
   while ( move = MoreMoves() ) {           //選一個走子
      MakeMove(move);                       //走步
      if( firstmove )
            v = - Nega_Scout( depth - 1, -beta, -alpha );
      else{
            v = - Nega_Scout( depth - 1, -alpha-1 , -alpha );
            if( v > alpha && v < beta )
                  v = - Nega_Scout( depth - 1, -beta, -alpha );
      }
      UnmakeMove();                         //還原盤面
      if (v >= beta) {
         return beta;
      }
      if (v > alpha) {
         alpha = v;
      }
   }
   return alpha;
}
```

圖 9 Nega_Scout 演算法

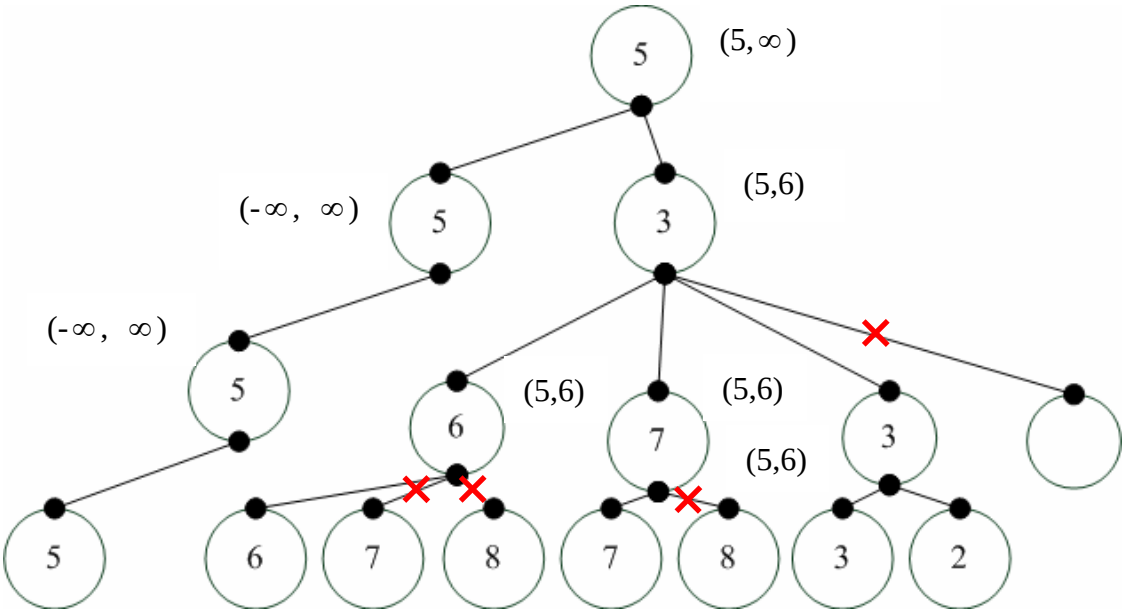

圖 10 Nega_Scout 遊戲樹

範例見圖 10。最左邊的Alpha_Beta window為(-∞，∞)，當搜尋到葉節點的時候，將分數回傳到根節點，根節點的值為 5，Alpha_Beta window就更新成為(5, ∞)，以下的Alpha_Beta window就是使用(5, 6)去搜尋。

由這個範例可以知道 Nega_Scout 演算法通常比 Alpha_Beta 更能省下時間。