棋形知識庫之設計與製作

DESIGN AND IMPLEMENTATION OF PATTERN KNOWLEDGE BASE

嚴祁麒
國立台灣大學資訊工程研究所博士生
私立僑光技術學院國際貿易系講師
jengchi@ocit.edu.tw

顏士淨
國立東華大學資訊工程系助理教授
sjyen@mail.ndhu.edu.tw

摘要

一盤由圍棋高手所下出來的棋，與一盤由初學者所下的棋，在盤面上一眼就可以分辨出來，原因就是高手下棋時非常講究棋子的形。在雙方進行短兵相接的攻防中，好的棋形會使得自己的局勢順暢，有利於後來發展。以專業棋手來說，在他們的腦中至少有上萬種棋形的知識。這種知識懂得愈多，對於棋力的提昇就有愈大的幫助。

不過要建立及儲存這些棋形的知識還需要一些技巧，在本論文中我們採用 5 ×5 固定大小的棋形樣式作為儲存單位，藉以建立一個棋形知識庫。這個知識庫不但可提供初學者學習圍棋，更可以進一步做為電腦圍棋程式的知識基礎來源，有助於目前電腦圍棋程式的發展。

關鍵詞：棋形、棋形知識庫、棋形樣式、圍棋、電腦圍棋。

ABSTRACT

It is easy to distinguish the GO games played by the masters from those played by the beginners, because GO masters emphasize the shapes of those moves very much. When the fight begins in the GO game, good shapes will lead the position to a better result in the future. In terms of a GO expert, there is knowledge of at least ten thousands of shapes in his brain. The more knowledge he has, the more help to enhance his rank of GO.

We need some useful techniques to create this knowledge and save it. In this paper we use a pattern with 5×5 fixed size as a storage unit, then create a pattern knowledge base by these patterns. This pattern knowledge base not only provides the beginners for learning GO, but could be taken for the source of knowledge base in Computer GO program. It is good for the development of Computer GO currently.

Keywords : shape, pattern knowledge base, pattern, GO, Computer GO

一、緒言

(一) 圍棋的相關知識與規則

圍棋是一種由黑白雙方在縱橫十九路的棋盤上，互爭地盤的遊戲。它是目前被公認為最難、最具有深度的棋類。圍棋的規則簡單，但變化極為複雜、玄妙深奧，在彼此的鬥智中充分展現了微妙的較勁場合。

在棋局進行中，會有某些棋子被對方吃掉的情形。在棋盤上如果有部分棋子是同色而且

相連接，則稱這部分的棋子為一個「棋串」。每個棋串都會有所謂的「氣」，其定義也就是棋串之棋子四周的空點數目總和。如果在下棋的過程中，有某個棋串因為被敵方包圍而氣數總和為 0，則這個棋串便被敵方吃掉了，我們必須將它從棋盤上移除。

當棋局進行到尾聲，再無寸土之地可爭時，棋局便告結束。此時計算雙方圍取的地域目數(圍到的 1 個空點稱為 1 目)，以圍取較多目數的一方為勝者。

(二) 棋形的概念

對於圍棋有相當認知的人都知道：一盤由高手所下出來的棋，與一盤由初學者所下的棋，在盤面上一眼就可以分辨出來。最大的原因就是高手下棋時非常講究棋子的「形」，而初學者往往還沒有形的觀念。因為圍棋進行的過程中，棋子是一顆一顆逐漸累積到盤面的。在雙方進行短兵相接的攻防中，好的棋形會使得自己的棋勢順暢，有利於後來發展；而不佳的棋形則使得棋子缺乏效率，甚至於留有缺陷，對於往後必定有不利影響。

所謂的棋形是指雙方棋子在棋盤上之局部區域所構成的形態。一般來講，黑白雙方在局部有所接觸時，不論是要攻擊或防守，要擴張己方或壓制對方，要追擊或者逃命，其實都有固定的某些常用手法可以使用。

例如圖 1 是一個在實戰中常出現的棋形，這時候輪到白棋。以正常情況來說，在這裏白 A 是一著好棋，其它的著手都不好。這對於有圍棋概念的人而言，可以說是一種棋形常識；以專業棋手來說，在他們的腦海中至少有成千上萬種棋形的知識。這種棋形的知識懂得愈多，對於棋力的提昇就有愈大的幫助。或者也可以說，想要下好圍棋，就一定要多研究棋形上的相關知識 [7][8]。

本篇論文主要在探討如何利用電腦來有效地建立一個棋形知識庫，使得我們可以透過這個知識庫來學習圍棋的棋形；並且針對建立知識庫的方法作進一步的研究。這個知識庫不但可提供初學者學習圍棋，更可以進一步做為電腦圍棋程式的知識基礎來源，有助於目前電腦圍棋程式的發展。

二、棋形的樣式與分類

(一) 棋形的樣式

根據上一章所討論，棋形的知識對於學習圍棋具有相當大的幫助。如果我們要嘗試建立一個棋形知識庫，首先必須要對於種種的棋形有一個明確的認識及觀念。當然這與圍棋的專業知識有關，所以棋形知識的歸納與整理，也是專家系統中的一門學問。

建立棋形的重點主要在於擷取資訊。所謂的「擷取」，就是要根據各種盤面上的棋子狀況，分析出那些是關鍵子，那些是較不重要的棋子，以利於我們整理出相關的棋形。以圖 1 為例，在此局面白棋下在 A 位長出是一種好形。可是再觀察圖 2 之情形，此圖雖然與圖 1 不全相同；但這時候白棋下在 A 位依然是好形，並且它的狀況與圖 1 多少也有點關係。

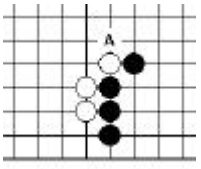


圖 1

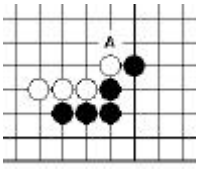


圖 2

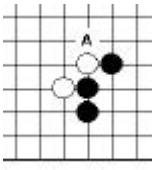


圖 3

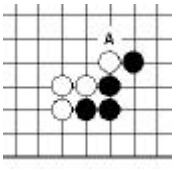


圖 4

經過這兩圖的仔細分析觀察，我們可以據以建立出圖 3 的棋形樣式(Pattern)出來，並且很明顯圖 3 的 A 點是白方的一個好點。如果我們將圖 3 的棋形樣式儲存起來，則當我們遇到了像是圖 4 的情形時，馬上就可以知道這時候白棋下在 A 點是好棋。

這種分析觀察與歸納，然後再擷取出關鍵棋子的功夫，是要藉由專業的知識來完成的。其中對於雙方棋子的相對位置、周圍環境之配合等等都有重要的關係，所以具有相當之難度。另一方面要注意的是，每種棋形樣式中的好點，都具有它相關的作用，例如擴張己方、壓制對方、整理己方棋形、襲擊對方弱點、逃出己方、追擊對方、連結己方、分斷對方等等，有時候一個棋形樣式的好點還同時具有數種作用。因為我們在思索每步棋該如何下時，應該都會有一些目標做為導向；所以根據既定的目標，再配合找出符合目標的棋形好點，這樣對於下出好棋或妙手是大有助益的。

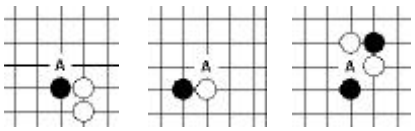
(二) 棋形的分類

如上一節所述，在製作棋形樣式時，必須要根據專業知識對每個棋形詳加推敲，經過歸納與分類整理之後，才能擴充到棋形知識庫中。所以這樣的工程是很浩大的。

以下我們針對建立棋形知識庫時，如何將棋形樣式進行分類的方法作一個探討：

(1) 依棋形術語分類

在圍棋的手法上，經常會根據雙方棋子配置關係而有對應的圍棋術語，例如長、扳、尖、跳、虎等等。這種分類方法對於一個懂棋的人來說，是最為理想的分類法，而且在建立棋形時較不會遺漏。圖 5 列出一小部分根據棋形術語所設計的棋形樣式，這些樣式的術語是暫訂以黑子為主。



(a) 長　(b) 扳　(c) 斷

圖 5

(2) 依棋形用途分類

這種分類方式是依照棋形的作用來分類，例如急所、整形、連結、壓迫、擴大等等。這種分類方式對於應用在電腦圍棋上較有幫助，因為在一個優秀的電腦圍棋程式中，必須要具備目的導向式(object oriented)的搜尋系統；也就是在搜尋時，必須要選擇能達成某些作用的著手。因此根據棋形用途來分類的做法，就能夠在需要時提供適合的著手。

但是這種分類法在實際建立知識庫時，就會出現不容易歸納完整的情況。第一是因為棋形樣式所指出的好點，其用途經常不只有一種而已。例如一個單純的一間跳，圖 6 的情形黑 A 跳是擴大陣地，但圖 7 的黑 A 跳則是逃命。另一個問題則是棋形樣式會發生重覆的機率很大，這在製作上會有若干的困擾。

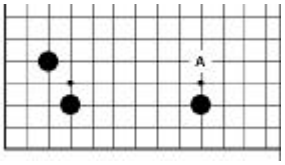


圖 6

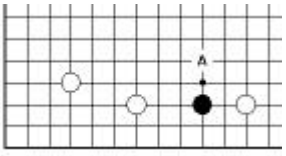


圖 7

經過權衡利弊得失，我們認為在製作過程仍以採用依圍棋術語分類的方式較好，最大的理由是知識庫的收集整理會比較完整，也比較容易進行歸納與偵錯。至於棋形的功能則可在整理歸納時同時加以設定，並不至於造成製作上的麻煩。

三、 棋形知識庫的建構模式

(一) 棋形樣式的製作

在本節中我們要把前面所提一個個的棋形樣式，透過電腦程式將它設計出來。首先我們要先決定每個棋形樣式的大小，為了製作上的便利，棋形樣式的大小最好是固定 $n \times n$ 的型式。當然 n 值愈大則歸納出來的棋形就愈細緻，考慮的內容也會愈完整；但從反面來說，

每個棋形樣式所佔的空間也愈大，而整個知識庫也會因為棋形分的較細微而使得棋形樣式的數量倍增。根據實際分析與整理棋形的經驗，我們認為 5 × 5 是合適的大小[3][4]。因為絕大多數的基本棋形，例如長、扳、跳、飛、衝、斷等等，都能夠被包含在 5 × 5 的範圍中；而其它需要較大範圍的棋形，幾乎都在棋盤邊上，且這些棋形的範圍大體上是狹長型的(例如 5 × 10)。因此若我們以 5 × 5 為基本單位的話，在碰到這種較大範圍的棋形時，就可以採用合併法來處理(見下一章)。以下我們就以這種 5 × 5 大小的棋形樣式來說明其製作方法(這裏所提出的製作方法，對於 n × n 的大小也完全適用)。

在設計每一個 5 × 5 的棋形樣式時，我們會對這個樣式的 25 個點清楚指出：那些點一定是我方棋子、那些點一定是敵方棋子、那些點必須是空點；除此之外，有時候某些點不能是我方棋子，有時候某些點不能是敵方棋子。又有時候某些點是無關緊要的，也就是說該點不論是我方或敵方棋子甚至是空點都無所謂，這樣的點我們稱為無關點(don't care point)。圖 8 是一個棋形樣式的設計範例，在這個例子中，O 代表我方棋子，X 代表敵方棋子，· 代表空點，&代表不能是我方棋子，@ 代表不能是敵方棋子，_ 則代表無關點[3]。

```
_ _ _ _ _
_ X O _ _          33  33  6
_ ·X _             4   8   00
_ &@ @ _
_ _ _ _ _
```

圖 8

光看圖 8 的棋形可能還無法充分理解這個棋形的意義，譬如說這個棋形所指出的好點在何處並未標示。所以對於每個棋形，我們還需要一些資訊來輔助。在圖 8 右邊我們設了幾個數值，分別代表了重要的屬性：

(1) 我方與敵方的好點之位置：如前面兩個座標 33 與 33，這代表此型不論輪到誰，都應該下到此型中心點(3,3)的位置上。

(2) 此棋形的用途代號：如數值 6。由於每個棋形的用途可能有好幾種，例如連接與分斷、封鎖與突圍、擴張與壓迫等等。因此我們事先整理好許多不同的用途代號，例如 1、2、4、8、16…等等(都是 2 的冪次以利於編碼)。所以數值 6 表示這個棋形兼具用途 2 與用途 4 的功能。

(3) 此棋形的重要程度：如數值 4。在這裏我們把棋形的重要程度分為四類，分別是緊急(4)、重要(3)、一般(2)、特殊狀況(1)。大致上具有連接與切斷功能的，其重要程度會比較高。這樣的分類對於日後進行搜尋時，能夠迅速提供較佳之好著點，有助於提昇搜尋的品質與效能。

(4) 此棋形需要的比對次數：如數值 8。因為每個棋形與實際的盤面作棋形辨識比對時，除了原形之外，通常需要再作 90 度、180 度、270 度等共 4 種角度的旋轉；另外看情況可能每種角度可能還要作鏡射(對稱)處理，因此最多需要比對 8 次，有時也會有只需要 4 次或者 2 次的情形。

(5) 在棋盤位置的限制：因為有些棋形適用的場合是在角上或邊上，所以我們的棋形樣式必需指明是否是這種情況。數值 00 的意思是代表這個棋形並沒有這方面的限制。

設計好棋形樣式之後，如何根據這個樣式來進行有效的比對工作是很重要的，我們將在下一節詳加說明。

(二) 棋形辨識的方法

當我們建構好棋形樣式的結構之後，剩下的問題便是如何將這個棋形儲存起來。為了要

使棋形辨識的速度可以達到較高的效率，我們利用位元比對(bit mapping)的觀念來製作[3][4][6]。在這個方法中，我們將原來棋形樣式的 25 個點給予編號如圖 9 所示，每個編號將會分別對應到一個 32 位元整數中的一個位元。在棋形樣式中的 6 種代號 O、X、.、&、@、_ 分別用 6 個整數加以儲存，每個整數中所對應的位元如果有該代號則值為 1，否則其值為 0。

25 24 23 22 21
20 19 18 17 16
15 14 13 12 11
10 9 8 7 6
5 4 3 2 1

棋形樣式編號

□□□□□□□□□□□□□□□□□□□□□□□□□

32313029282726252 4 ..... 5 4 3 2 1

對應的整數位元編號

圖 9

以圖 10 的棋形樣式為例，在該樣式裏每個代號所儲存的整數，其內容如圖 12 所示(在此以二進位表示，且為了方便只寫出後面 25 個位元)，這個已儲存好的棋形樣式的整數內容是固定不變的。現在讓我們考慮到實際的棋局狀態，我們會利用 3 個 19 ×19 的整數二維陣列(假定取名為 B、W、E)，來分別記錄以黑子、白子以及空點為主的整數內容。譬如說整數 B[x][y]之內容，代表在以座標(x, y)為中心的 5 × 5 之範圍內，每個點如果是黑子則所對應之位元就將它設成 1，否則就設為 0 (位置對應參照圖 9)。

_ _ _ _ _
_ _ O . _
_ X . & &
_ _ O & _
_ _ _ _ _

圖 10

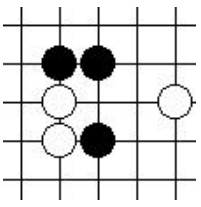


圖 11

代號 O 之值= 00000 00100 00000 00100 00000
代號 X 之值= 00000 00000 01000 00000 00000
代號 . 之值= 00000 00010 00100 00000 00000
代號& 之值= 00000 00000 00011 00010 00000
代號@ 之值= 00000 00000 00000 00000 00000
代號_ 之值= 11111 11001 10000 11001 11111

圖 12

B[x][y] = 00000 01100 00000 00100 00000
W[x][y] = 00000 00000 01001 01000 00000
E[x][y] = 11111 10011 10110 10011 11111

圖 13

現在如果某個盤面其中有個 5 ×5 的範圍如圖 11 所示，則對於圖 11 的中心點(x, y)而言，其所儲存的三個陣列內容分別如圖 13 所示。

以圖 10 的棋形樣式對照圖 11 的盤面，可以知道它們是吻合的。由棋形樣式所儲存的 6 種代號的整數值，與實際盤面資料所得的 B、W、E 三個陣列元素的整數值，我們設計出一個效率很高的 Bitmapping 演算法得知這個盤面與棋形樣式是否比對成功。這個演算法主要是利用 AND、OR、NOT 三種位元邏輯運算來完成，其內容如下：

Algorithm Bitmapping：

Input ： ① 棋形樣式之 Integer O, X, P, &, @, D (P, D 分別表示空點 .與無關點_ 所存之整數值)；② 實際盤面之 Integer B, W, E。

Step 1：令 Integer D' = NOT D (將 D 的每個位元變補數)
Integer K' = NOT & (將&的每個位元變補數)
Integer A' = NOT @ (將@的每個位元變補數)

Step 2：if ((**D'** AND **A'** AND **B** = **O**) 且 (**D'** AND **K'** AND **W** = **X**) 且 (**D'** AND **A'** AND **K'**

AND **E** = **P**))

then

RETURN SUCCESS

Step 3： if ((**D'** AND **A'** AND **W** = **O**) 且 (**D'** AND **K'** AND **B** = **X**) 且 (**D'** AND **A'** AND **K'** AND **E** = **P**))

then

RETURN SUCCESS

Step 4： RETURN FAILURE

在這個演算法中，如果 Step 2 **的條件成立，則表示用** O **代替黑子、用** X **代替白子的比對成功。如果** Step 3 **的條件成立，則表示用** O **代替白子、用** X **代替黑子的比對成功。若** Step 2 **與** Step 3 **都不成功則比對失敗！**

現在拿上面圖 12 **的六種棋形代號值, 與圖** 13 **實際盤面計算所得的** B, W, E **之值來執行看看：**

Step 1 ： D' = NOT D =

00000 00110 01111 00110 00000

K' = NOT & =

11111 11111 11100 11101 11111

A' = NOT @ =

11111 11111 11111 11111 11111

Step 2 ：D' 00000 00110 01111 00110 00000

A' 11111 11111 11111 11111 11111

And B 00000 0 1100 00000 00100 00000

O 00000 00100 00000 00100 00000

(**條件 D'** AND **A'** AND **B** = **O 滿足**)

D' 00000 00110 01111 00110 00000

K' 11111 11111 11100 11101 11111

And W 00000 00000 0 1001 01000 00000

X 00000 00000 0 1000 00000 00000

(條件 D' AND **K'** AND **W** = **X 滿足)**

D' 00000 00110 01111 00110 00000

A' 11111 11111 11111 11111 11111

K' 11111 11111 11100 11101 11111

And E 11111 10011 1 0110 10011 11111

P 00000 00010 00100 00000 00000

(條件 D' AND **A'** AND **K'** AND **E** = **P 滿足)**

故可知比對成功。

使用這個演算法進行棋形辨識是很快的。在前置作業上只要事先將知識庫中的棋形樣式讀入並且設定好，並且準備 3 **個** 19 ×19 **的二維陣列, 來儲存棋盤中黑白子與空點的位元編碼資訊。接著棋盤中只要每下一個子，我們就立刻修改二維陣列裏對應的編碼值, 隨後便立即可以查出棋形樣式的辨識是否成功。對於每個棋形可能有旋轉、對稱等** 8 **個不同方向的變化，我們可以另外撰寫轉換程式加以處理, 或者多花一些記憶體空間去儲存棋形樣式的** 8 **種變化值即可。**

(三) 其它的棋形知識庫製作方式

在這裏我們要討論另一種棋形知識庫的製作方式。這種方式的特點是每個棋形樣式的大小是不固定的，而形狀也不一定是矩形，它可以是任意的不規則形, 例如圖 14 **的**(a)**與**(b)**,**

(a) (b)

2 7 5 5 5

1 * 4 2 1 1 1 2

4 2 1 1 * 1 1 1

8

連接與切斷 一線上尖渡

圖 14

這種表示法的編碼是透過了特別的安排。在圖 14 **中是分別用數字『**1、2、4、8**』來代表『空點、我方棋子、敵方棋子、邊界之**

外』四種情形；其它數字則代表兩種以上的可能組合。例如 5 表示該點可以是空點(數字 1)或是敵方棋子(數字 4)，7 表示該點無論是空點(1)、我方棋子(2)或是敵方棋子(4)都可以。而＊則表示在這種棋形樣式裏應該要下的位置。

在過去採用這種實作方式的電腦圍棋程式，包括了劉東岳的 Dragon 2.0 與顏士淨的 Jimmy 5.0 [1][2]。很明顯地，這種方式由於大小不固定，因此對於每個棋形樣式而言比較有彈性，而且利用 1、2、4、8 進行的組合式編碼設定也很靈活。但是它卻有幾個不利的缺點存在：

(1) 在棋形編輯與檢視時，它的可讀性很低，不像 O、X 那樣清楚易懂。所以日後擴大整個知識庫規模時，整理歸納與編修工作會很麻煩。

(2) 這個方法的棋形樣式看來大小雖小(多數不需要 5 ×5 =25 個點)，然而其儲存空間卻並不節省。這主要是因為它的棋形樣式形狀不固定，所以儲存每個棋形時，也必須把它的「形狀」儲存起來。譬如它可能要儲存每個點的組合編碼以及該點與＊點的相對位置，這有如 sparse matrix 的儲存方法，所以此法就儲存空間來說並不經濟。

(3) 更重要的一點是，這種方式在與真實盤面進行棋形比對時；對每個棋形樣式來說，必須要針對該棋形樣式中每一個『點』分別進行檢查。在這一方面上它無法像前述方法一樣使用 bitmapping 的技巧，因此在速度上而言亦有相當之落差。

所以經過研判，我們認為應該放棄這種作法。因為它的儲存空間與比對時間都遜於固定大小式的作法。特別是日後要作搜尋處理時，若要將棋形知識庫建議的著點提供給搜尋系統作為選擇，那麼棋形比對的效率就變得極為重要了，因此我們認為固定大小的作法是較佳的選擇。

四、 特殊用途的棋形處理

本章我們要對一些具有特殊用途的棋形加以討論，並研究其處理方法。包括大型棋形樣式之處理方式，以及棋形中棋子的狀況問題兩類。

(一) 大型的棋形樣式之處理

在第三章我們選定以 5 ×5 做為棋形樣式的大小，這是一種經過利弊權衡下的決定。當然這樣的大小範圍，對於大多數的棋形來說都可以滿足所求；不過在一些特定場合，特別是在棋盤的邊上，往往會有一些涵蓋範圍比較大的棋形，例如圖 15 所示。

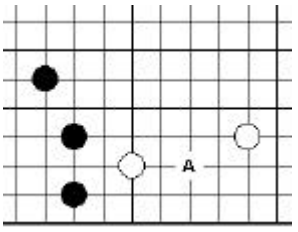


圖 15

遇到這樣的棋形，如果再加大 5 ×5 的範圍(譬如說用 9 ×9) 並不是明智的選擇，因為這徒然只是為了滿足少數 case 而大增使用之記憶體空間，而且執行速度也會受到拖累而變慢。在這裏我們採用的解決方式是允許棋形樣式合併(merge)，也就是可以讓兩個 5 ×5 的棋形樣式合併在一起變成 5 ×10 的範圍，例如圖 15 的狀況我們可以用圖 16 的兩個 5 ×5 之棋形樣式來表示。

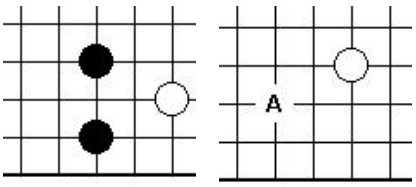


圖 16

採用合併法最大的好處，在於原來的資料儲存方式以及棋形辨識方法完全不必更改，我們只須在要作合併處理的棋形樣式裏加上辨識標記，再略作一些改變就可以成功處理。因此，透過合併法我們可以有效解決大型棋形樣式的問題。

(二) 特殊棋子狀況之處理

有時候棋形樣式指出的好點，應用在實際盤面時反而不好。這是因為在棋形辨識時，只是找出它們的「形態」是否一致，但並沒有詳加考慮棋子的「狀況」。譬如圖 17 的棋形樣式是用來找出切斷的好點，如果黑棋用根據它來下在圖 18(a)盤面上的 A 點，那就是一步嚴厲的好棋；但如果在圖 18(b)的情況，則黑 A 的切斷就變成是自找死路之著，因為下一手棋被白棋下在 B 位，則黑子會被提取。

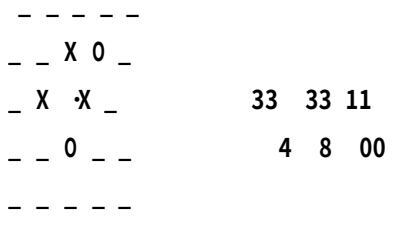
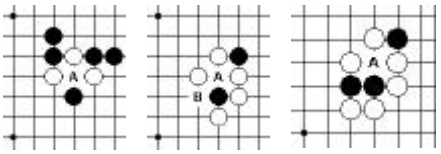


圖 17



(a) (b) (c)

圖 18

在這裏我們也不能因為這個原因而把圖 18(b)作成惡形之棋形樣式去避開不下，那樣並不能解決問題，因為類似的情形可能會出現極多種，例如圖 18(c)即是。

圖 18(a)至(c)的例子之所以會出現這樣大的差別，其關鍵在於我們沒有將黑子的「氣數」考慮進去。也就是在圖 17 之樣式中，位於中心空點下方的 O 子，其氣數必須不少於 2 氣才不會判斷失誤。

除了氣數之外，類似的誤判情形也可能會發生在死活有問題的棋子身上。例如圖 20 來說，黑 A 的切斷事實上毫無意義，因為黑方的△兩子已經是死棋狀態了，所以 A 位的切斷雖然是個棋形辨識上的要點，但實際上卻沒有切斷的效果，可以說是辨識上的盲點。

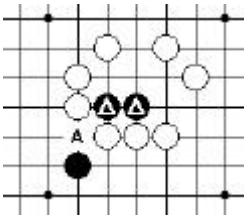


圖 20

要避免這種因棋子狀況而造成的誤判，基本上有兩個解決辦法。其一就是在棋形樣式中附加一些額外的條件，像是某棋子必須在多少氣以上(或以下)，以及棋子是否要求必須是安全的等等。但是這樣的作法有相當大的壞處：第一是製作棋形知識庫時要耗費過多的精力在許多難以預料的例外處理上，因為每個不同的棋形樣式要求的棋子狀況會有差異，甚至同一個棋形裏也可能會有數組不同的狀況需求(例如要求 O 的氣數必須不小於 X 的氣數)。第二是會加重棋形辨識上的負擔，不但速度上會大幅降低 bitmapping 的優點，而且棋形知識庫也會變得複雜，不易於日後的維護與管理。

所以我們認為比較理想的解決辦法，就是不在這個地方作處理！因為從一個模組化的角度來思考，每個不同的模組(module)應該各有它不同的功能，也各有它該做與不該做的事。如果我們要進行電腦圍棋程式的設計，則棋形知識庫最多只是提供可能的好點或指出要避開的惡形，至於棋子的氣數與安危狀況應該是由更上層的決策模組去綜合考量。也就是說，棋形知識庫的指示是提供一個參考，這個參考是否正確，必須要交由上層的決策分析系統去評估後方能得知；如果直接要在棋形知識庫裏就要做好這樣的判斷，那是事倍功半而且

也會造成電腦圍棋整體結構層次的紊亂。

五、結論

經過對棋形的整理歸納與製作方法的研究，我們認為：

1. 採用以圍棋術語進行棋形的分類是較佳的方式。這樣做不但整理時較不會遺漏，而且作法上比較有系統。另外，每個棋形各有其一種或多種用途，這點在處理上充分可以做到。
2. 棋形樣式的設計上宜採用固定大小模式，這樣可以利用位元比對的技巧加快棋形辨識的速度。而且因為格式的統一，使得棋形知識庫的維護與修改比較容易。
3. 在處理特殊棋形時，例如大型棋形樣式、或者因為棋子狀況而造成的誤判等等。要把持的基本原則是『絕不能因此而把整體結構改變而複雜化』，在處理上不但要力求簡明，而且要仔細思考每種模組應該完成的功能範圍為何，才不至於走錯大方向。

我們將這些設計概念與方法實際撰寫出程式，再透過一些對局實戰譜作為測試範例來進行測試。經實驗結果，在速度上如我們所料的相當理想；而在正確性上也相當令人滿意，所有的棋形要點都能確實找出。

然而在實作過程上，我們發現最難的部分還是在於棋形的整理歸納。第一因為這是需要專業的知識，棋力水平不夠則無法做好。再則是即使有相當的專業素養，在製作過程也會有思慮上的疏失；因為這項工作本身資料量就很龐大，而且資料與資料之間存在有相當的重複性(因為棋形樣式之間具有「無關點」的組合)，偶爾也會充斥著矛盾性，所以這是一項相當難且需要不斷測試修正的工作。

所幸整個知識庫的架構統一而且大致完整，且目前的測試修正已經趨於穩定，這點相當有利於日後的發展維護。並且以這樣的棋形知識庫系統，不論是作為學棋的工具，或者進一步將它用在電腦圍棋程式上，都有極佳的發展空間與價值。在計算機科學中的 Computional Complexity 的領域裏指出：圍棋的問題複雜度是一種 PSPACE 的領域(NP ⊆ PSPACE) [5]。我們相信若是對於圍棋之棋形辨識能進展到更高明的水準，必定有助於日後電腦圍棋這方面的研究，預計未來會有更好的發展。

參 考 文 獻

[1] 劉東岳，“電腦圍棋程式之設計與製作”，臺灣大學資訊工程研究所碩士論文，1989

[2] 顏士淨，“電腦圍棋程式 Jimmy 5.0 之設計與製作”，臺灣大學資訊工程研究所博士論文，1999

[3] 嚴祁麒，“電腦圍棋程式 Archmage 1.0 之設計與製作”，臺灣大學資訊工程研究所碩士論文，1992

[4] Mark Boon, “A Pattern Matcher for Goliath”, Computer Go, Winter 1989-90, No.13, pp.12-24

[5] Papadimitriou, “Computional Complexity”, Addison Wesley, 1994

[6] P. W. Prey, “Machine Problem Solving, Part 2: Directed Search Using Cryptarithmetic”, BYTE, Oct, 1980, pp.266-272

[7] Walter Reitman and Bruce Wilcox, “Pattern recognition and pattern-directed inference in a program for playing Go.” , Pattern-Directed Inference Systems, pages 503-523, 1978

[8] Richard J. Lorentz, “Pattern matching in a Go Playing Program”, Game programming workshop in Japan, pages 167-174, 1995