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

八層三角殺棋的勝負問題之研究

On the Study of 8 Layer Triangular Nim

研究生：白聖群　撰

中華民國九十八年六月

摘要

電腦棋類遊戲在人工智慧領域中是很重要的。而三角殺棋部份，自從許舜欽教授在 1985 年研究出七層三角殺棋結果後。三角殺棋更多層數的結果，就沒有任何相關文獻了。

在本論文中，我們使用 CPU 為 AMD Athlon64 X2 4000+ 2.1GHz，記憶體為 8G Byte 的個人電腦，花費約十四個小時半的時間，證明了兩種規則的八層三角殺棋皆為先手勝。

我們除了找到八層三角殺棋的結果，也提出了一些解三角殺棋勝負時，可以加快搜尋勝負速度的方法。雖然研究過程中花費許多時間在倒推法上，但我們也研究出來所有先前求出的盤面是可以運用到之後要求解的三角殺棋。並且提出了一個管理記憶體的方式，使得在求解三角殺棋的過程中，盤面資訊狀態可以儲存，這樣就可以利用較少量記憶體來解八層三角殺棋，而不必動用到虛擬記憶體。

關鍵字：人工智慧、三角殺棋

ABSTRACT

Computer chess games are very important in the field of artificial intelligence. There is no research results on Triangular Nim in higher dimensions Since Professor Shun-Chin Hsu solved 7 layer triangular Nim in 1985.

In this thesis, a personal computer equipped with AMD Athlon64 X2 4000+ 2.1GHz CPU and 8 GBytes RAM is utilized to conduct our experiments. Thus, it spends 14.5 hours and gets the results that two kinds of 8 layer Triangular Nim are a first-player win.

In addition, we find some skills for improving the performance of our programs. We can save the sub-problem results and reuse these results for solving Triangular Nim in higher dimensions. A memory management scheme is also proposed to reduce the memory requirements which can avoid the virtual memory swapping.

Keywords: artificial intelligence, Triangular Nim

目 錄

附表目錄

附圖目錄

第一章　諸論

第一節　前言

三角殺棋相傳最古早的淵源來自於中國，當時的人們利用小石頭的堆砌，進而有規律的發展出這麼一套有趣遊戲。輾轉流傳到現代，其遊戲方式算是 Nim 遊戲[1][2]的一種變形。Alan Tucker 在「Applied Combinatorics」[3]一書也曾經提到三層和四層的三角殺棋。在坊間，三角殺棋亦有一些遊戲網站[4]提供線上服務供人玩此遊戲。

三角殺棋是兩人進行的益智遊戲，一開始的遊戲棋盤是排成正三角形，正三角形的棋盤高度即為此三角殺棋之層數。如圖 1-1 的盤面，此盤面即為八層三角殺棋。

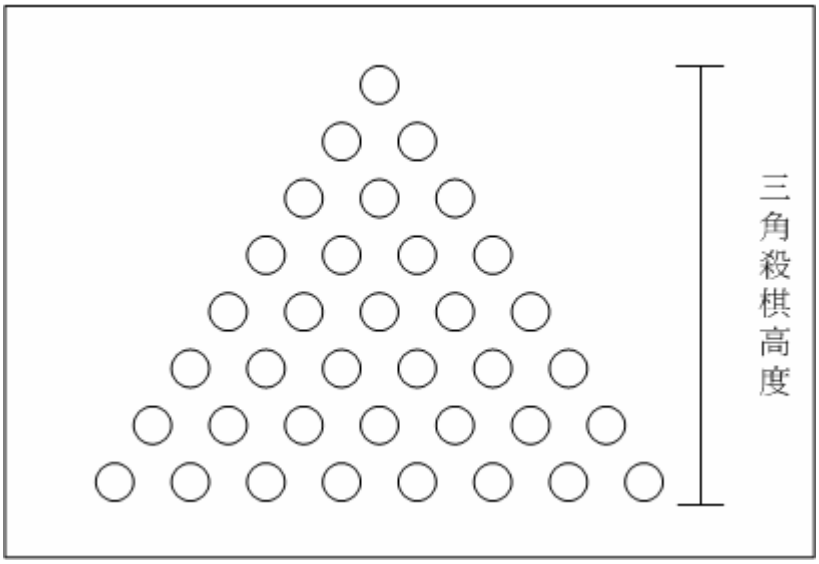
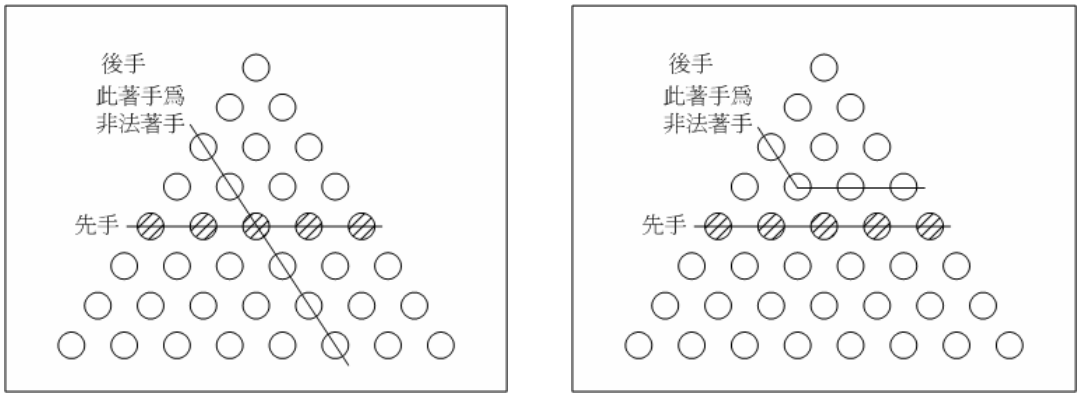


圖 1-1 八層三角殺棋

遊戲規則主要是有兩種，一種是從棋盤取得最後一顆子獲勝，另一種是從棋盤取得最後一顆子為失敗。每次玩家選取的棋子，在取子的時候不限定遊戲者取

子數目，但必須相連且成同方向的一直線，而且必須至少取一子。

下棋過程中，棋子的狀態只有被劃過與沒劃過兩種。在取子過程中，若該棋子的狀態已經為劃過，則不能再改變其狀態。如圖 1-2 (a)，劃過走過的棋子為非法著手。另一種情況，如圖 1-2 (b)，沒有相連成同一方向也是非法著手。遊戲進行中，這些都是不被允許的。遊戲進行中，任一方也不能選擇 pass 放棄著手。



(a) 著手劃到已經劃過的棋子　　(b) 著手沒有相連成同一方向

圖 1-2 非法著手

決定好玩家取子先後次序，雙方再不斷地輪流取子，總有一個時刻棋子會被取光，雙方皆無步可走時，那時就是決定勝負之時，如圖 1-3。而此遊戲只有贏或輸，並沒有和局這種狀況。

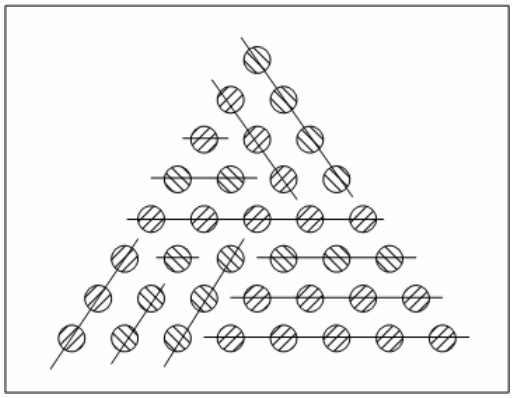


圖 1-3 雙方無步可走

一般來說在三角殺棋的遊戲中，人類與電腦程式對弈是不可能會獲勝的，這是因為電腦能比人類往前看更多的三角殺棋走法變化，從一開始下棋開始，便先設想到許多步棋以後的棋局變化，在第一步棋就能選擇最有優勢的候選步，規劃出無懈可擊的策略。然而以七層三角殺棋來說，可能的盤面組合實在是太多了，所有的盤面組合的數目是 2^{28}，計算複雜度也相對提高。

截至目前為止，我們已經知道在規則為取得最後一子為敗的情況下，二、四、六、七層三角殺棋皆會是先下必勝，一、三、五層則是先下必敗[5][6]。因而推估八層三角殺棋應該也是對先手極為有利的遊戲。

第二節　研究動機

我們知道三角殺棋算是 Nim 遊戲的一種變形，而在白啟光的數學嘉年華網站[7]有提到，Charles Leonard Bouton 在本世紀初利用了二進位的方式，解開了 Nim 這個遊戲的法則。

既然 Nim 能夠找到其規律性的結果，我們相信類似 Nim 且有著固定規則的三角殺棋，或許能夠找到一個規則來解開這遊戲的法則。而我們在探索及嘗試各種演算法增進破解三角殺棋程式的速度時，所發現的方法，也許也能用在其他棋類遊戲，從而在電腦博奕領域中貢獻綿薄之力。

第三節　論文架構

本論文共分為五章。第一章為諸論，簡述前言、研究動機及論文架構。第二章介紹電腦三角殺棋發展之歷史，以及一些相關研究。並敘述其使用的基礎理論，如對局樹的展開、倒推法等程式設計技巧。第三章為嘗試解八層三角殺棋的一些過程及演算法設計理念等。第四章為原先演算法的一些改進策略。第五章為結論及未來研究方向。

第二章　相關文獻及基礎理論

第一節　相關研究成果

以下我們介紹一些具有代表性的電腦殺棋相關研究成果。

1.以倒推法證明三角殺棋勝負

在許舜欽教授的論文中[6][7]，因為該遊戲難以用傳統的分析方式解決，故直接使用倒推法來解該問題。也就是說，並不使用估值函數及 MIN-MAX 搜尋，或配上 $\alpha-\beta$ 切捨。倒推法主要利用了電腦快速運算及大量記憶的特性，將遊戲的所有可能狀況全部計算出來，找到最好的應手。

而許舜欽教授利用倒推法解開了一到七層三角殺棋的勝負問題，其三角殺棋規則為取到最後一子為敗。勝負情形如表 2-1。

三角殺棋層數	勝負情形
一層	先手負
二層	先手勝
三層	先手負
四層	先手勝
五層	先手負
六層	先手勝
七層	先手勝

表 2-1 三角殺棋勝負情形(取得最後一子為敗)

許舜欽教授使用的機器為 VAX-11/750，記憶體為 32M Bytes，共花費三天半的時間求出七層三角殺棋為先手勝的結果。

2.三角殺棋的經驗智慧(heuristic)

巫光楨在尤怪之家之三角棋解析中[8]，將三角殺棋的一些子盤面作完整的剖析，將 3、4、5 枚餘子相連但未形成一直線的殘型尋找之間的等價關係，舉例來說，在規則為取得最後一子為勝的情況下。若盤面狀態為三枚餘子相連但未形成一直線，如圖 2-1。該子盤面有何特性呢？解析如下：

先手若取 A ，則後手可取 BC 得勝。

先手若取 B ，則後手可取 AC 得勝。

先手若取 C ，則後手可取 AB 得勝。

先手若取 AB ，則後手可取 C 得勝。

先手若取 AC ，則後手可取 B 得勝。

先手若取 BC ，則後手可取 A 得勝。

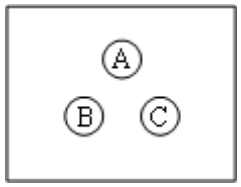


圖 2-1 三枚餘子相連但未形成一直線

由以上可以推導出，圖 2-1 由 ABC 表示的盤面和空盤面是一樣的。因為在規

則為取得最後一子為勝的情況下，後手勝的盤面搭配先手勝的盤面，必為先手勝。所以若有兩個盤面，一個盤面分析必為後手勝，則該盤面和空盤面等價，不會影響整個盤面結果，如此只要尋找另一個盤面的勝負。在圖 2-2 的情況之下，圖中兩個盤面為等價的，皆是先手勝。

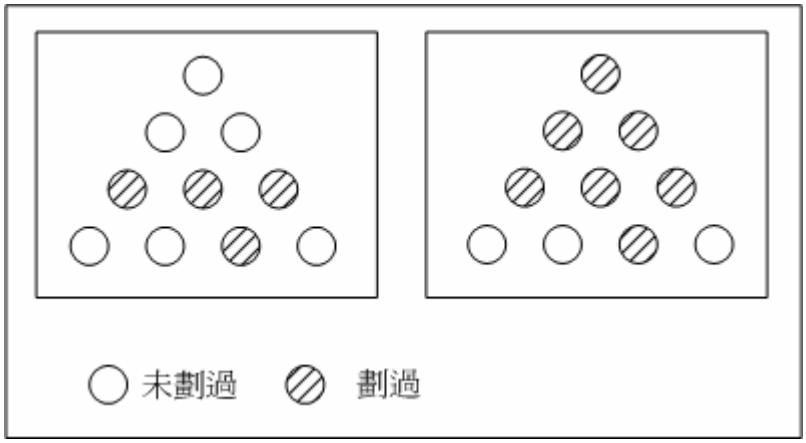


圖 2-2 兩個等價盤面

藉由這樣的分析，去分析 3、4、5 枚餘子的盤面，進而推廣到更多棋子的子盤面，指導遊戲玩家能充分利用該子盤面的勝負情形，使玩家能處於最有利的狀況之下來下棋。

第二節　Game Tree 直接展開

這是一般解遊戲勝負最常用的方法，像是在象棋、暗棋、黑白棋及其他對奕遊戲都常用此方法。假設輪我方著手，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應對等，如將這一連串的對奕過程以樹狀結構表現出來，可以稱此樹狀結構為對局

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變化都搜尋一遍的，因為需搜尋的盤面數量是十分龐大的，故通常皆會使用棋盤審局函數來減少搜尋的節點數量。

但本研究在解八層三角殺棋勝負問題時，我們不考慮這個方法。主要原因是遊戲樹展開時的分支度實在是太大了，在二層三角殺棋時遊戲樹分支度只有 6 個。但隨著層數繼續累加到八層三角殺棋，Game Tree 的分支度將成長到 288 個。Game Tree 這樣的成長速度對於記憶體使用以及程式撰寫上都是一大難題。

此方法僅對於較小層數的三角殺棋適用。除非能找到有效的修剪對局樹方法以及較好的棋盤審局函數，否則使用此方法很難在合理時間內解八層的三角殺棋。所以我們無法像在解暗棋或黑白棋等遊戲[9][10]，使用 Game Tree 展開並使用該遊戲的經驗法則來有效降低 Game Tree 的分支度。

第三節　倒推法

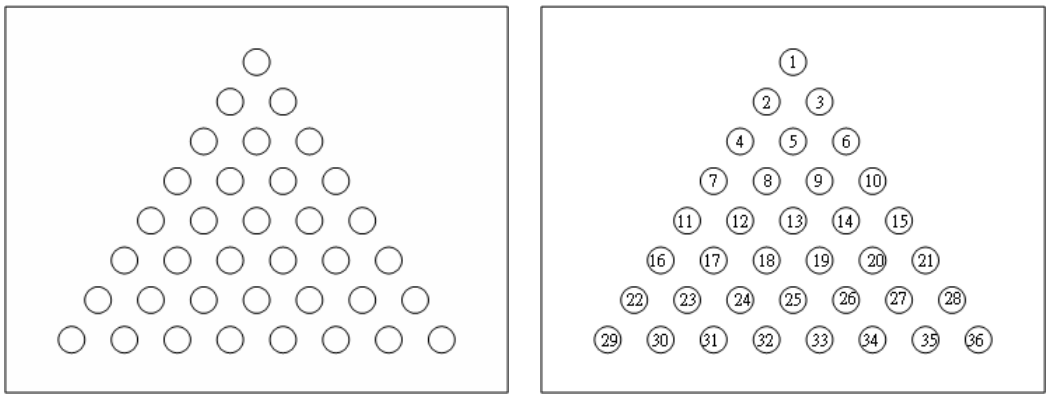
倒推法是許舜欽教授在「利用電腦研究七層三角殺棋的勝負問題」論文[7]中所提到的演算法，該演算法的時間複雜度為 O($2n^{(n+1)/2} * n^2$ (n+1)/2)，其中 n 為三角殺棋的層數。因此，隨著層數的增加，所花費的計算時間及記憶體容量的成長是相當驚人的。

以八層三角殺棋來說，倒推法就必須利用 2^{36} 個位元(8G bytes)來記錄每一盤面狀態的勝負情況。若某狀態走一步後的下一個狀態均是先手必勝，則此狀態

為先手必敗。因為以此狀態來說，不論接下來考慮的著手為何，由此狀態到下一狀態皆是先手勝，皆找不到致勝路徑。所以此狀態會被設為先手敗。反之，只要下一個狀態有一個是先手敗，則可以推導出目前狀態為先手勝。

由於三角殺棋的盤面狀態只有「未劃過」或「劃過」兩種狀態，只要一個 bit 即可明確表示。因此，解八層三角殺棋，我們可以利用 36 個位元來表示 2^{36} 個種盤面狀態。而八層三角殺棋所有狀態可對應到整數 0 ~ 整數 $2^{36}-1$。

首先是將棋子排成如圖 2-3(a)的三角形，並且將八層三角殺棋的棋子編號如圖 2-3(b):



（a）編排成三角形棋盤　　（b）三角殺棋編碼盤面

圖 2-3 三角殺棋編碼

編碼完的三角殺棋對應到 36 bit 整數的方式就如圖 2-4，每一個 bit 為 0 或 1。

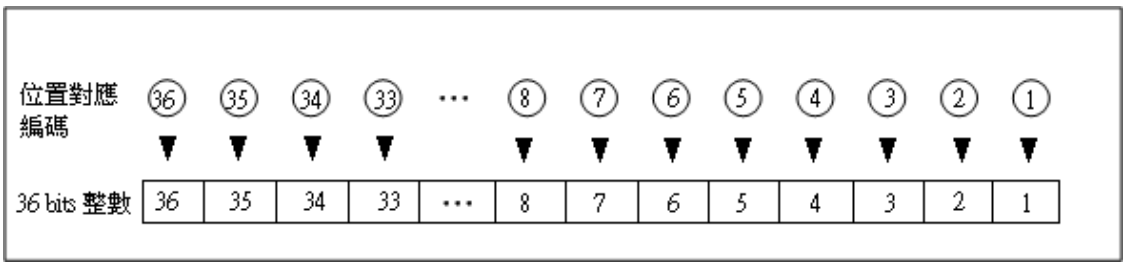


圖 2-4 位置對應編碼示意圖

編號 2^{36}-1 代表棋盤上全部棋子皆被劃過，然後由此狀態的勝負倒推到編號 0，也就是倒推到棋盤上所有棋子皆存在的起始狀態，便可以決定三角殺棋的勝負。這個演算法最早是許舜欽教授提出[7]，如圖 2-5。

```
S(2^36-1)= 必勝 ;

for(i=2^36 - 2;i>=0;i--)
{
   S(i) 爲必敗 ;
   for(j=1;j<=288;j++)
   {
       if( 第 j 著手爲合法 )
       {
           求出 S(i) 下第 j 著手之後的盤面狀態 ;
           if( 下完後的盤面狀態爲必敗 )
           {
               S(i) 爲必勝 ;
               break;
           }
       }
   }
}
```

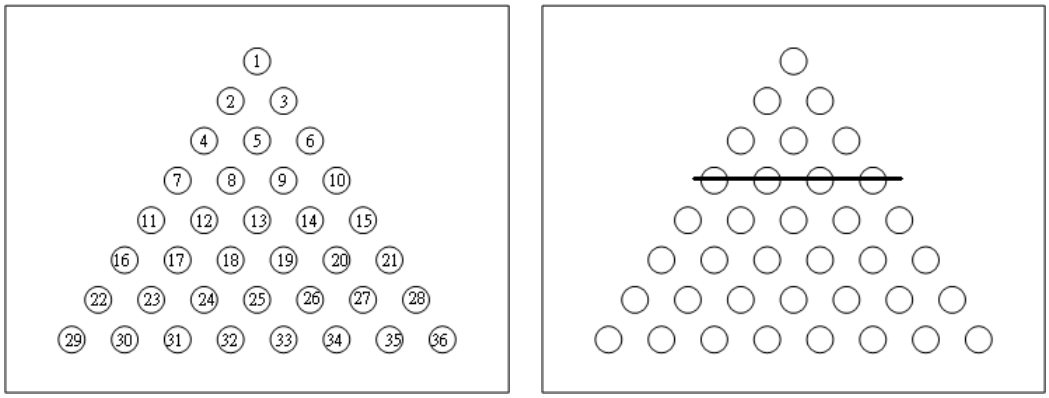
圖 2-5 倒推法演算法

圖 2-5 是解規則為劃掉最後一個棋子為負的倒推法。若要將規則改為劃掉最後一個棋子為勝，我們只需將一開始時 $S(2^{36}-1)$的狀態設為必敗即可。因為根據演算法一開始的設計原則，目前狀態是由以前狀態推導出來。所以當 $S(2^{36}-1)$設為必敗時，也就是所有棋子被劃完的前一個狀態皆為勝，也就符合了劃掉最後一個棋子為勝的規則。

第四節　可行著手集合的產生

在使用倒推法之前，必須先建立八層三角殺棋的可行著手集合。可行著手集合的建立，也是延續許舜欽教授在「利用電腦研究七層三角殺棋的勝負問題」中所提到的建立方法。

可行著手集合的建立方法，是根據我們在倒推法一開始建立好的編碼盤面，如圖 2-6(a)。然後根據資料結構的狀態。如圖 2-6(b)可行著手劃掉了棋盤上的 7、8、9、10，再根據圖 2-7 的示意圖，我們可以知道 bit7、bit8、bit9、bit10 皆為 1，其他 bit 皆為 0，這樣的著手編碼後二進位值為 00000000000000000000000001111000000，轉成整數值為 960。而利用這樣的編碼方式可將所有可行著手的值給尋找出來。



（a）三角棋編碼盤面　　（b）可行著手一例

圖 2-6 三角殺棋著手編碼示意圖

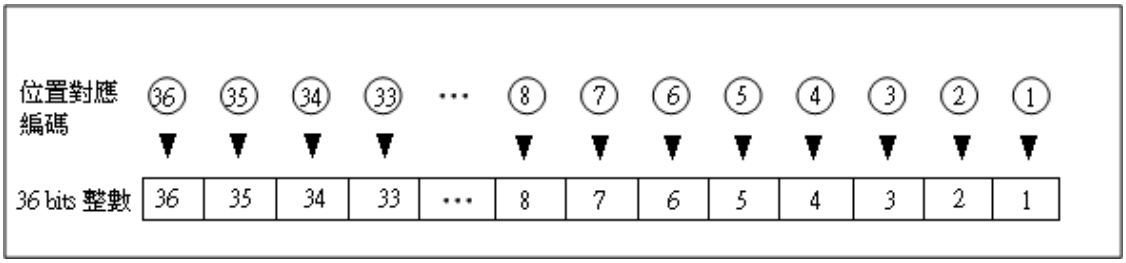


圖 2-7 位置對應編碼示意圖

然後我們建立一個棋盤上每個棋子之間的位置關係圖，針對棋子的右邊、左下、右下建立一個連結表格。因為三角殺棋有這三個方向的可行著手，我們建立這個表格的目的，就是為了檢查各方向是否還有相鄰的棋子。建立的連結表格如表 2-2。其中 0 表示該方向並沒有相鄰的棋子。

位置	右邊	左下	右下
1	0	2	3
2	3	4	5
3	0	5	6
4	5	7	8
5	6	8	9
6	0	9	10

7	8	11	12
8	9	12	13
9	10	13	14
10	0	14	15
11	12	16	17
12	13	17	18
13	14	18	19
14	15	19	20
15	0	20	21
16	17	22	23
17	18	23	24
18	19	24	25
19	20	25	26
20	21	26	27
21	0	27	28
22	29	29	30
23	24	30	31
24	25	31	32
25	26	32	33
26	27	33	34
27	28	34	35
28	0	35	36
29	30	0	0
30	31	0	0
31	32	0	0
32	33	0	0
33	34	0	0
34	35	0	0
35	36	0	0
36	0	0	0

表 2-2 八層三角殺棋位置連結表格

有了位置連結表格，我們就可以根據圖 2-8 的演算法(也是由許舜欽教授提出)，依序由每一個點開始，產生經過該點的右邊方向、左下方向及右下方向的

所有可行著手的值求出來，以便和三角殺棋的倒推法合併使用。

```
n 值爲可行著手的值，Link 爲連結表

for ( i = 1 ; i <= 36 ; i ++ )
{
    n = 2 (i-1) ;
    輸出 n 值 ;
    for ( 方向 = 右邊，左下，右下 )
    {
        n = 2 (i-1) ;
        j = i ;
        for ( k = 1 ; k <= 7 ; k ++ )
        {
            if ( Link ( j , 方向) 等於 0 )
            {
                此方向結束
            }
            else
            {
                j = Link ( j , 方向);
                n = n + 2 (j-1);
                輸出 n 值 ;
            }
        }
    }
}
```

圖 2-8 輸出所有可行著手的演算法

舉例來說，若現在要求取位置為 2 出發的所有可行著手，首先會輸出 $2^{(2-1)}$也就是 2^{1}。內層的 for 迴圈則是求取會經過位置 2 的右邊方向、左下方向及右下方向的可行著手。然後查詢連結表格，表格內位置 2 右邊方向的值是 3 不為 0，就輸出 $2^{1}+2^{(3-1)}$，同時將 j 值更新為 3。接下來查連結表時，就是查 3 的右邊方向的值是否為 0。根據這樣的方式，將所有可行著手全部輸出。

第三章　如何解八層三角殺棋？

第一節　Divide-and-Conquer

在本研究過程中所使用的演算法與資料結構，是延續許舜欽教授在「利用電腦研究七層三角殺棋的勝負問題」論文中所提到的演算法[7]，其演算法名稱為倒推法。在我們這個研究中也是使用倒推法來判斷八層三角殺棋的勝負。

但如果直接使用許舜欽教授提出的演算法，不對盤面狀態壓縮，則需要記錄 2^{36}= 64G 個盤面的狀態，若每個狀態用 1 byte 記錄，則需要 64G Bytes 的記憶體，這樣的記憶體需求實在是太大了。若盤面狀態壓縮也需要 8G Byte 的記憶體，這樣則必會動用到系統中的虛擬記憶體。在電腦記憶體的考量之下，我們試圖找尋新策略以及改進原有方法來解八層三角殺棋勝負。

在初期的研究中，是預期使用 Divide-and-Conquer 的概念下去尋找三角殺棋勝負結果，將大問題切割成一些小問題，先去計算小問題的解，並儲存結果以供稍後的計算使用，然後建構成整個問題的解。在這裡我們是切割為兩個子問題。

為了尋找八層三角殺棋勝負的方便性，我們首先考慮將遊戲規則變更為劃掉最後一個棋子為勝，也就是三角殺棋在前言提到的另一個規則。這點與許舜欽教授的論文有些許不同，該論文中是以劃掉最後一個棋子為負。所以，一層到七層的結論必須重新尋找，但有鑒於搜尋空間遠小於八層，此變更方式仍然是值得。而且也把另外一種遊戲規則的三角殺棋結果給找出來，對學術上研究也是有幫助的。

在這裡之所以要變更遊戲規則的原因是，為了要運用 Divide-and-Conquer 的概念，故假設一開始第一個玩家就將整個三角殺棋盤面切割為兩個小區塊。如圖 3-1，一個小區塊是三角形，另一個小區塊是梯形。如此一來，只要能找到三角殺棋上面三角形盤面為先手必敗，下面梯型為先手必敗，則可證明出劃掉最後一子為勝的八層三角殺棋為先手必勝。

當然會有這樣的推測，主要是因為許舜欽教授在劃掉最後一子為負的三角殺棋中，一、三、五層的結論是為先手必敗。而第六、第七層卻是先手必勝，這樣似乎可以歸納出越多層，對先手越有利。

若是不變更遊戲規則的情況下，也就是取得最後一子為敗，我們下完一個著手將三角殺棋分割為兩個子盤面時，在兩個三角殺棋子盤面中，我們對於兩個子盤面，皆是後手的狀態。在這樣的情況下，會產生三個 Case。

Case 1：若切割完的兩個子盤面皆為後手勝。

Case 2：若切割完的子盤面皆為先手勝的情況。

Case 3：若切割完的子盤面一個為先手勝、一個為後手勝。

因為在下棋過程中，雙方皆可自由地在兩個子盤面進行著手，所以無法有效保證各子盤面的勝負狀況。而且又因為每次取子的策略不固定，在這樣的規則之下，雙方有時會在盤面搶最後一顆子，有時又會避免取到最後一顆子。這樣增加了分析上的困難。

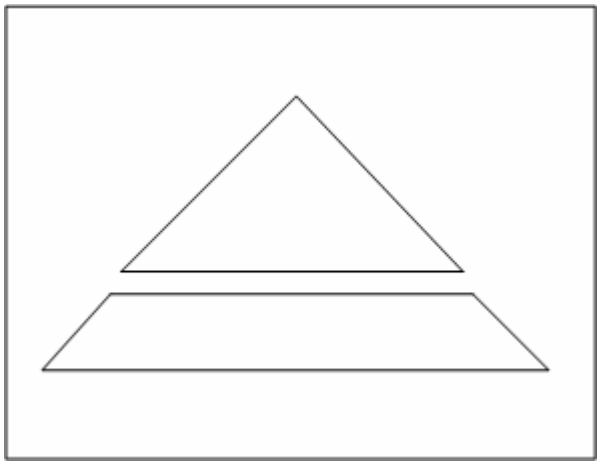


圖 3-1 Divide-and-Conquer 示意圖

但為了遊戲分析方便，我們無法預先規劃圖 3-1 三角形或梯形哪一個為先手勝，哪一個為後手勝，因為對手並非會按照我們意願，下在我們指定的地方。所以我們在研究初期就先考慮將規則換為取得最後一個子的玩家為獲勝。如此一來，在分析盤面資料時，我們一定可以確保無論是哪一個子盤面，不會因為先後手的關係，影響到勝負。

在變更遊戲規則的情況下（也就是取得最後一子為勝），我們下完一個著手將三角殺棋分割為兩個子盤面時，在兩個三角殺棋子盤面中，我們對於兩個子盤面，皆是後手的狀態。在這樣的情況下，也同樣會有三個 case。

Case 1：若切割完的兩個子盤面皆為後手勝。

Case 2：若切割完的兩個子盤面皆為先手勝。

Case 3：若切割完的子盤面一個為先手勝、一個為後手勝。

在這三個 Case 的情況下，我們可以如下證明 Case 1 必為後手勝，Case 3 必為先手勝，而 Case 2 則無法那麼直觀地去分析。

Case 1：若切割完的兩個子盤面皆為後手勝。在這種情況之下若玩家一為先手，玩家二為後手，這邊假設兩個後手勝的子盤面為盤面一、盤面二。

玩家一先到盤面一去下一著手，所以在盤面一為先手。此時玩家二最佳著手會是在盤面一與玩家一纏鬥。若玩家一與玩家二纏鬥至最後，則可以推導出玩家二必取得盤面一最後一顆子。該盤面為後手勝，所以玩家二必勝。於是到盤面二時，玩家二狀態會是後手。因為取完盤面一最後一顆子後，玩家一必然要在盤面二下任意著手。此刻玩家一為先手，推得該盤面必敗，因為盤面二後手勝。

若玩家一不與玩家二纏鬥，盤面一尚未結束就跳到盤面二，此時玩家一為先手，玩家二最佳著手為跟著玩家一到盤面二與玩家一纏鬥，若纏鬥至最後，則必然可以取得盤面二最後一顆子。所以，在這個規則之下遇到此兩種盤面組合，玩家二最佳下法就是只要玩家一跳離盤面，也跟著跳離該盤面，無論如何要與之纏鬥。則可以保證一定會取得兩盤面的最後一顆子，也就是取得這樣盤面組合的勝利。

Case 3：若切割完的子盤面一個為先手勝、一個為後手勝。在這種情況之下若玩家一為先手，玩家二為後手，這邊假設先手勝的子盤面為盤面一、後手勝的子盤面為盤面二。我們已經知道，兩個子盤面若後手勝，這樣可以推倒出整體盤面為後手勝。 所以，只要玩家一下一子到盤面一，使盤面一為後手勝盤面，則對玩家一來說整體盤面為後手，則必可取得整體盤面的勝利。也就證明了切割完的子盤面，若一個為先手勝、一個為後手勝必為先手勝。

藉由這樣的方式推導我們也可以證明，若一個先手勝盤面搭配無限多個先手敗盤面必為先手勝。

但是我們將一到七層的勝負結果，使用程式分析完成之後，發現結果並不如預期一開始猜想那樣，可以將分割結果拿來使用的並不多。而三角殺棋的勝負如表 3-1，根據三角殺棋一到七層的勝負表，僅有二層三角殺棋的結果是可以使用的。

三角殺棋層數	勝負情形
一層	先手勝
二層	先手負
三層	先手勝
四層	先手勝
五層	先手勝
六層	先手勝
七層	先手勝

表 3-1 三角殺棋一至七層勝負表(取得最後一子為勝)

根據二層三角殺棋的結果，我們對八層三角殺棋中剩餘的梯型盤面作重新編碼，並且產生新的可行著手數，而編碼的方式如圖 3-2。以此編碼方式所需的空間為 2^{30}bytes = 1G bytes，產生的可行著手數共有 210 種。在三角殺棋編碼的過

程中，最上面的三層是不編碼的。

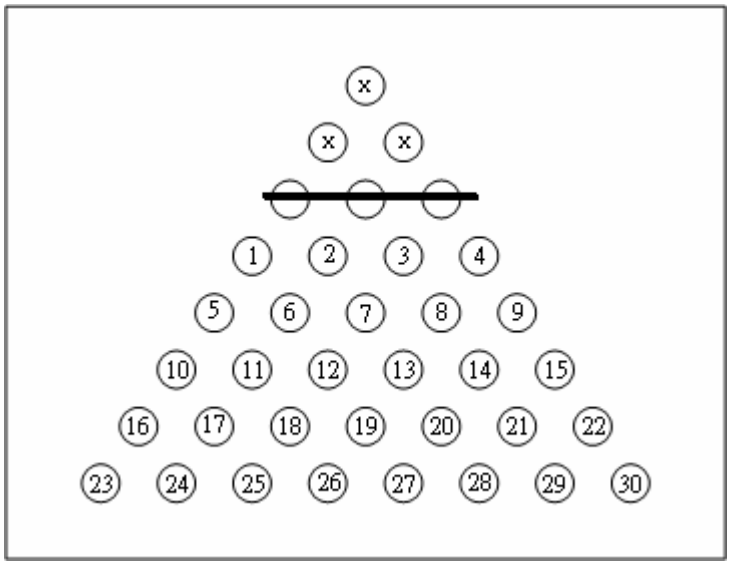


圖 3-2 三角殺棋梯形盤面編碼

但很可惜的是以這種方式來求解，並沒有達到預期目標。梯形方塊的勝負結果為先手勝，如此一來這樣的結果無法與二層的先手負結果相搭配，並無法推得八層三角殺棋為先手勝，而先手勝加先手敗的盤面資訊無法推論出任何結果，只能說這個方法失效。

第二節　搜尋必勝著手

雖然上一節將三角殺棋盤面以分割的方式來求勝負是失敗的，但在求解的過程中也引發了新的契機。在求三角殺棋勝負解的時候皆有把各盤面資訊記錄下來，因此想藉由觀察三角殺棋三到七層的第一手下在盤面中的何處才會獲勝，進而推導出八層三角殺棋盤面中，第一手最有可能會落在何處。

三層三角殺棋（規則為下到最後一子為勝）搜尋必勝著手結果如圖 3-3（每

個線段皆為一個著手)，雖然必勝可行著手數有六步，但其中有四步著手皆等價，故三層三角殺棋有兩個必勝著手，如圖 3-4。

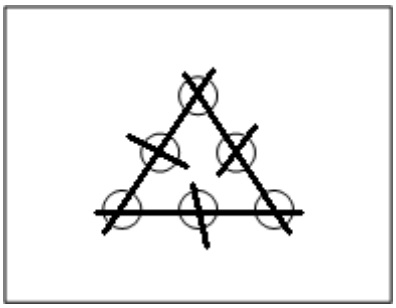


圖 3-3 三層三角殺棋第一步必勝著手

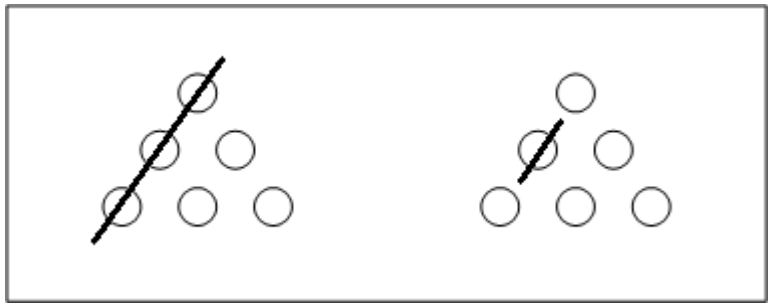


圖 3-4 三層三角殺棋第一步必勝著手

四層三角殺棋（規則為下到最後一子為勝）搜尋必勝著手結果如圖 3-5（每個線段皆為一個著手)，雖然必勝著手有六步，但嚴格來說該六步著手皆為等價，故四層三角殺棋只有一個必勝著手，如圖 3-6。

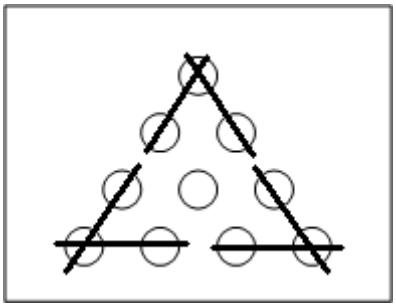


圖 3-5 四層三角殺棋第一步必勝著手

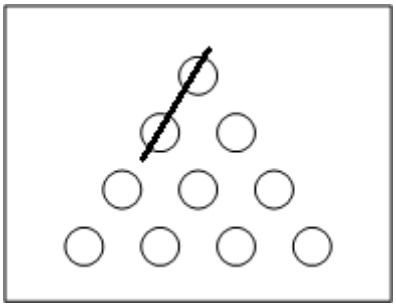


圖 3-6 四層三角殺棋第一步必勝著手

五層三角殺棋（規則為下到最後一子為勝）搜尋必勝著手結果如圖 3-7，雖然必勝著手有六步，但其中有四步著手為等價，故五層三角殺棋有兩個必勝著手，如圖 3-8。

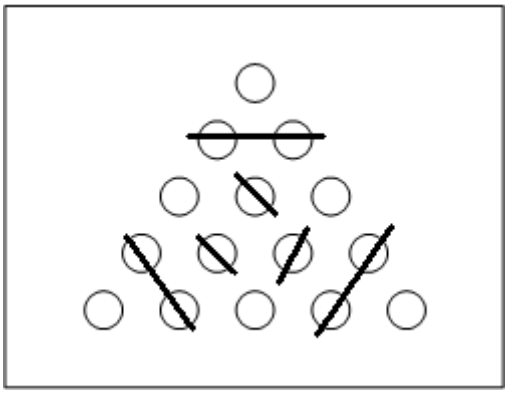


圖 3-7 五層三角殺棋第一步必勝著手

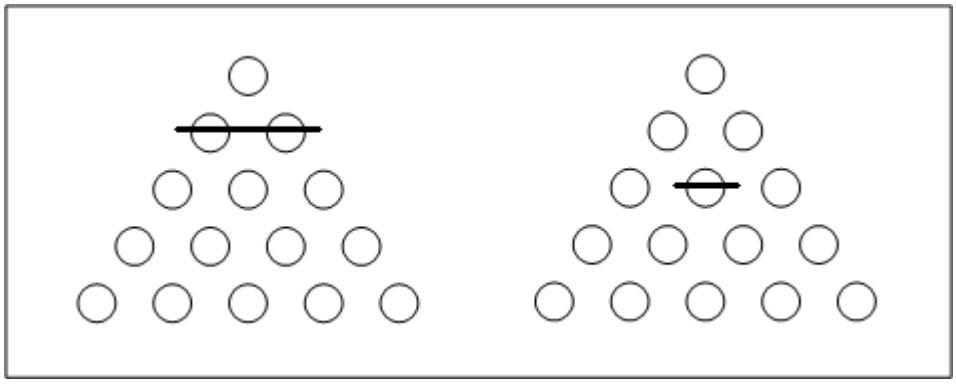


圖 3-8 五層三角殺棋第一步必勝著手

六層三角殺棋（規則為下到最後一子為勝）搜尋必勝著手結果如圖 3-9，雖然必勝著手有三步，但其中有兩步著手皆為等價，故六層三角殺棋只有一個必勝著手，如圖 3-10。

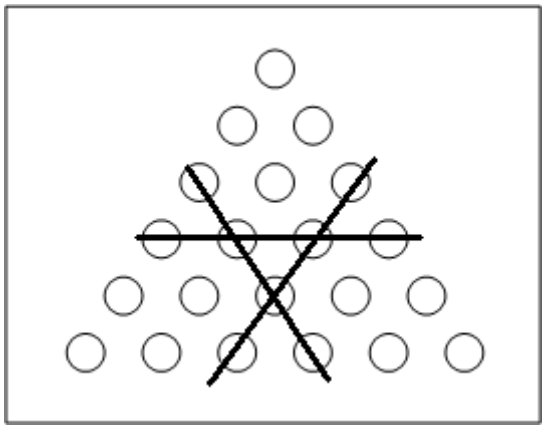


圖 3-9 六層三角殺棋第一步必勝著手

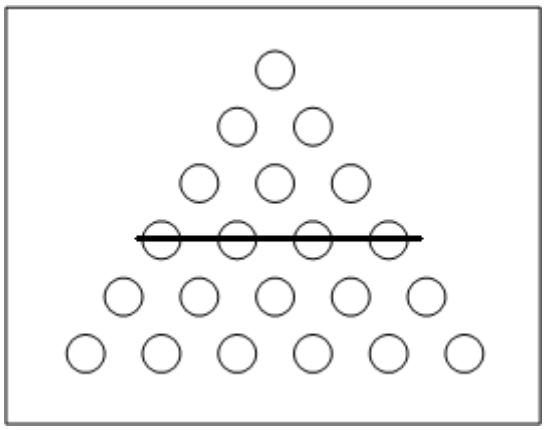


圖 3-10 六層三角殺棋第一步必勝著手

七層三角殺棋（規則為下到最後一子為勝）搜尋必勝著手結果如圖 3-11，雖然必勝著手有十步，但其中有六步著手為等價，故七層三角殺棋有四個必勝著手，如圖 3-12。

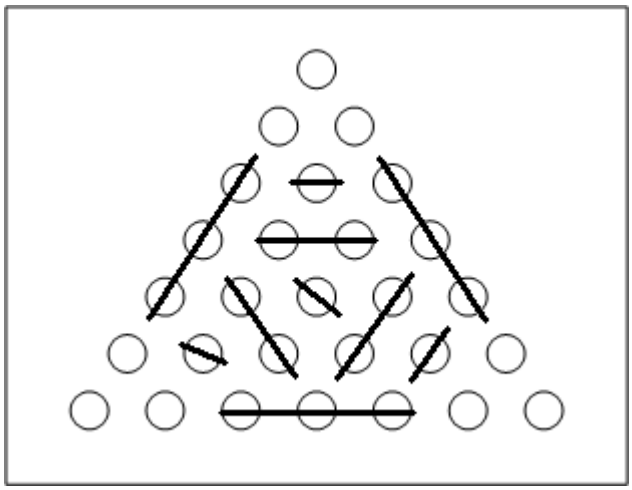


圖 3-11 七層三角殺棋第一步必勝著手

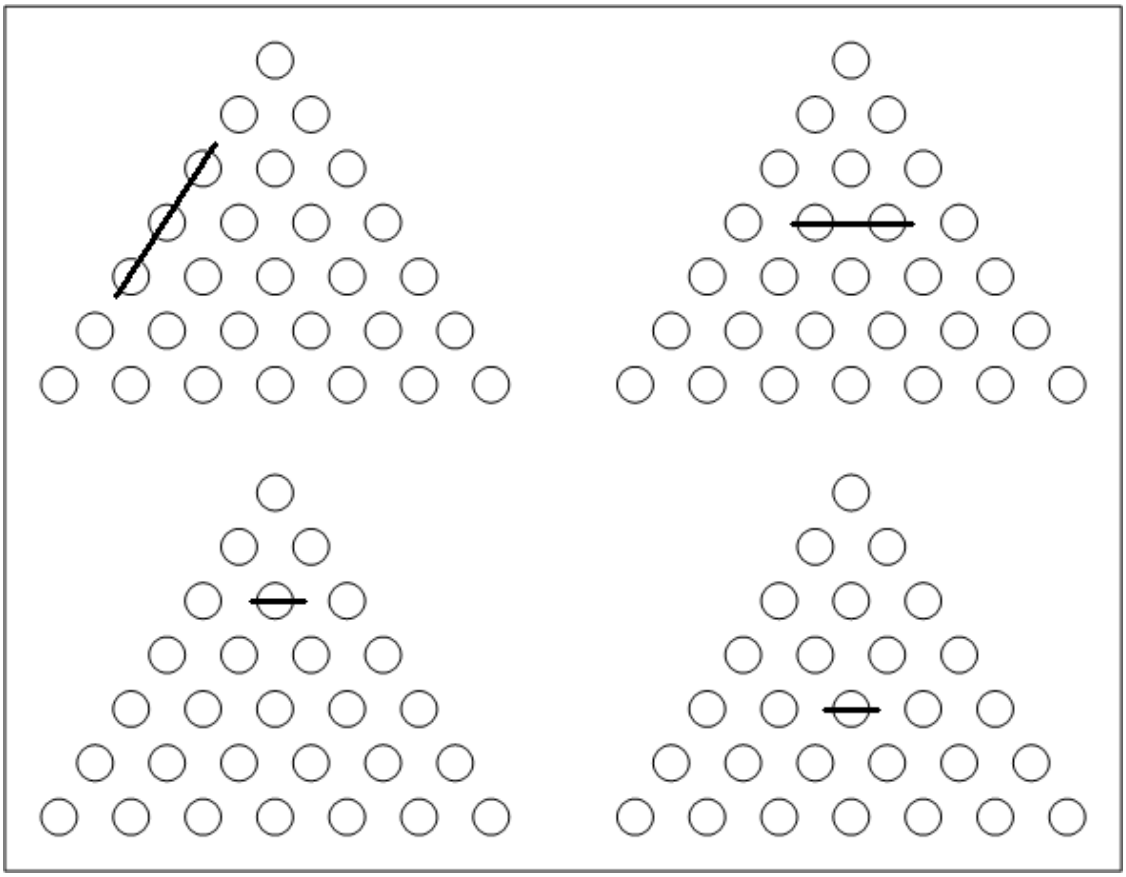


圖 3-12 七層三角殺棋第一步必勝著手

在規則為下到最後一子為敗的情況下，如表 2-1，先手必勝的盤面只有二、四、六及七層，二層的三角殺棋因為較簡單在此暫不討論，這個規則之下的盤面我們僅討論四、六及七層的必勝著手。

四層三角殺棋（規則為下到最後一子為敗）搜尋必勝著手結果如圖 3-13（每個線段皆為一個著手），雖然第一步必勝著手有三步，但嚴格來說該三步著手為等價，故四層三角殺棋只有一個必勝著手，如圖 3-14。

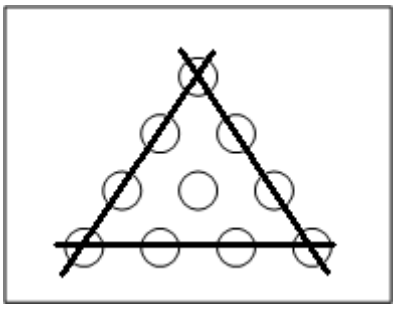


圖 3-13 四層三角殺棋第一步必勝著手

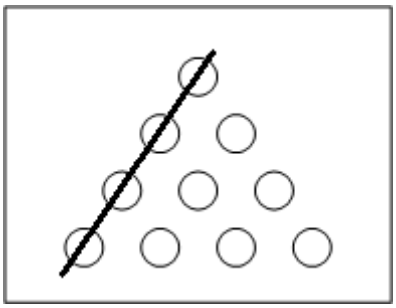


圖 3-14 四層三角殺棋第一步必勝著手

六層三角殺棋（規則為下到最後一子為敗）搜尋必勝著手結果如圖 3-15（每個線段皆為一個著手），雖然第一步必勝著手有十二步，但有八步著手為等價，故六層三角殺棋有四個必勝著手，如圖 3-16。

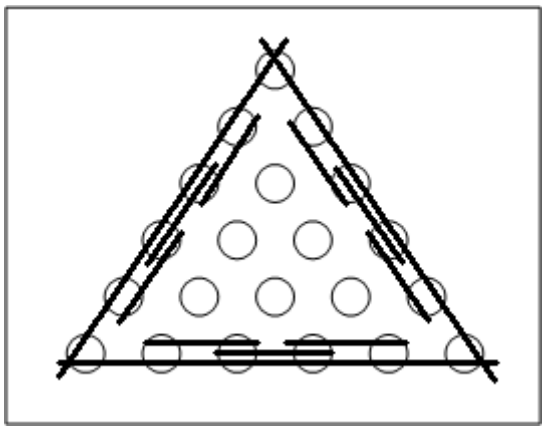


圖 3-15 六層三角殺棋第一步必勝著手

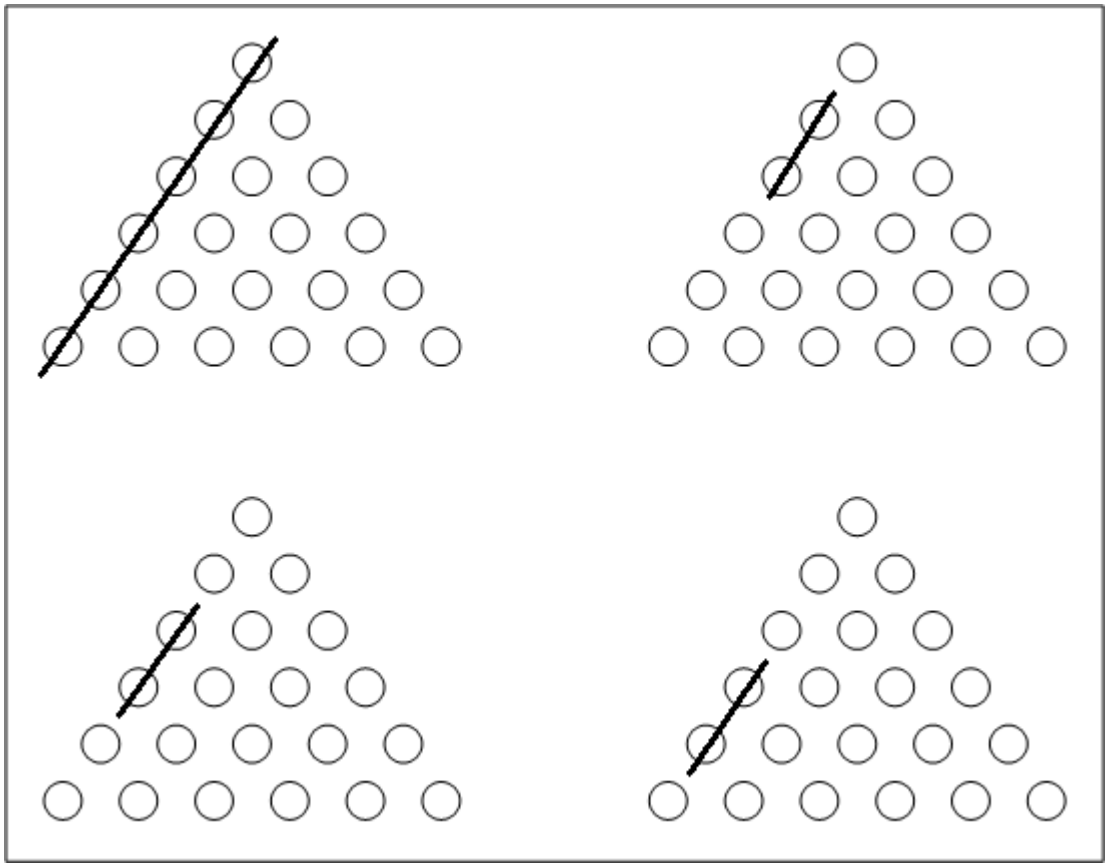


圖 3-16 六層三角殺棋第一步必勝著手

七層三角殺棋（規則為下到最後一子為敗）搜尋必勝著手結果如圖 3-17（每個線段皆為一個著手），雖然第一步必勝著手有六步，但有四步著手為等價，故七層三角殺棋有兩個必勝著手，如圖 3-18。

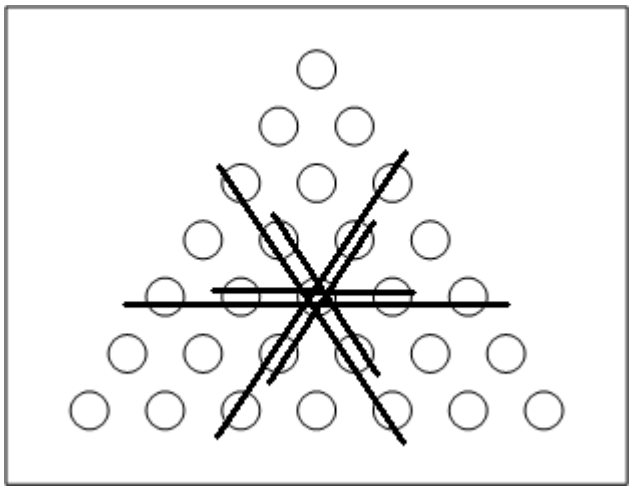


圖 3-17 七層三角殺棋第一步必勝著手

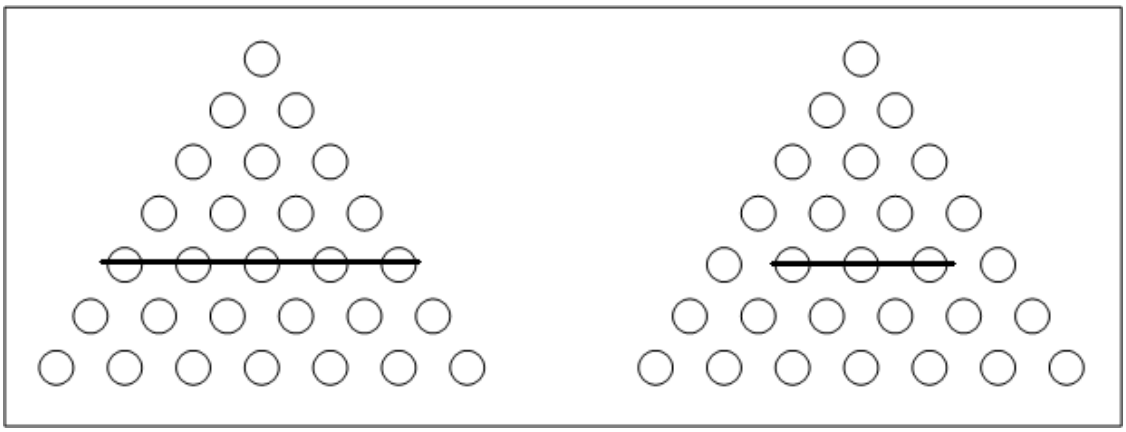


圖 3-18 七層三角殺棋第一步必勝著手

觀察了各層三角殺棋兩個規則下的第一手，我們首先考慮第一手著手有可能是將三角殺棋切割為兩個盤面的一刀。在規則為下到最後一子為勝的情況下，五層三角殺棋與六層三角殺棋皆有一著手是將三角殺棋切割為兩個盤面。在規則為下到最後一子為敗的情況下，七層三角殺棋有一著手將三角殺棋切割為兩個盤面。有了這樣的考慮為前提，我們又重新設計了八層三角殺棋的編碼，並且對八層三角殺棋進行七種測試，希望在這七種測試之下能夠找到必勝的第一著手。而這七種八層三角殺棋編碼表如圖 3-19。

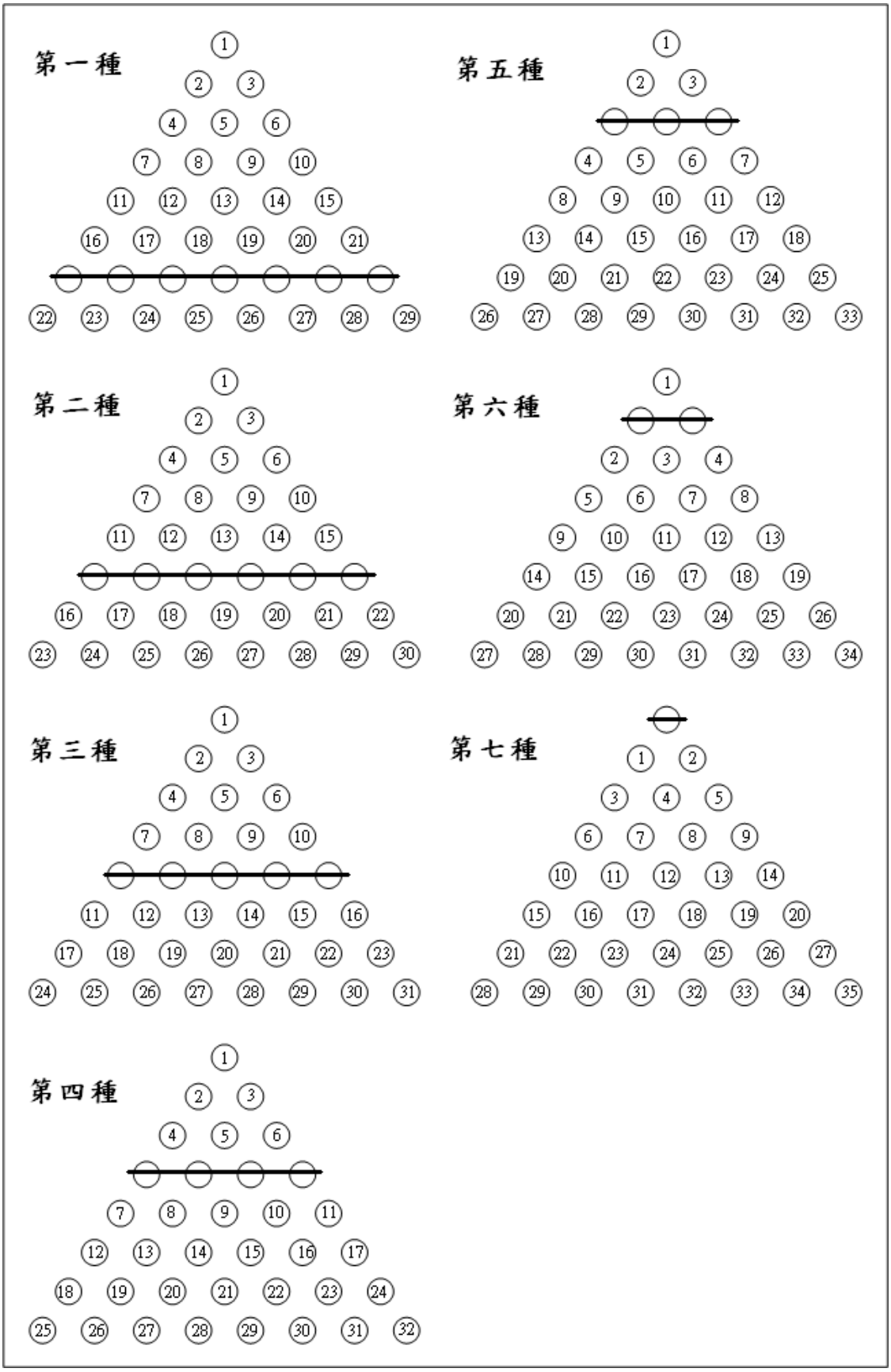


圖 3-19 七種八層三角殺棋編碼表

八層三角殺棋重新編碼後，每種切割法的可行著手數與所執行時所需要的記憶體如表 3-2。重新編碼後就利用許舜欽教授的倒推法求解，但由於進行研究時，所使用的個人電腦僅配有 8G bytes 的記憶體，故在第五種以上的八層三角殺棋必須進行盤面狀態的壓縮，由原先的 1 byte 存一個盤面資訊改為 1 bit 存一個資訊，如此一來即可將這七種測試完成。由於編譯器並未提供 1 bit 的資料型態，所以在下一節，我們會介紹壓縮資料型態的演算法。

	可行著手數	所需要的記憶體
第一種	163 種	約 2^{29} bytes（512M bytes）
第二種	153 種	約 2^{30} bytes（1G bytes）
第三種	154 種	約 2^{31} bytes（2G bytes）
第四種	186 種	約 2^{32} bytes（4G bytes）
第五種	216 種	約 2^{33} bytes（8G bytes）
第六種	247 種	約 2^{34} bytes（16G bytes）
第七種	273 種	約 2^{35} bytes（32G bytes）

表 3-2 三角殺棋可行著手數及所需記憶體

八層三角殺棋其實還有第八種測試可以做，但該測試其實並沒有必要做，該測試如圖 3-20，主要是第八種測試因為砍掉了最底層一刀，使得剩下的盤面等價

於七層三角殺棋。而七層三角殺棋的勝負結果已經知道是先手勝，搭配了這一種測試，我們可以知道會讓整個盤面等於先手敗。所以第八種三角殺棋測試是多餘的，並沒有辦法讓我們獲得想要的先手勝結果。

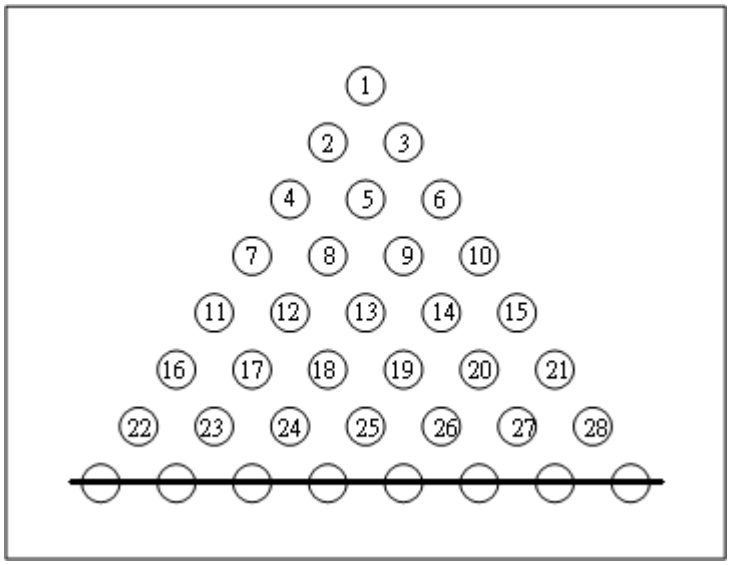


圖 3-20 多餘的八層三角殺棋測試

但在未改變盤面狀態的資料結構之前(也就是做前四種測試)，利用這個方法我們已經順利找到在規則為下到最後一子為敗的三角殺棋勝負結果。該規則下的八層三角殺棋如同預期般是先手勝，該勝負資訊是在第二種測試方法下找到，耗費的記憶體約為 1G Byte 左右，故可以知道該規則下的八層三角殺棋至少有一個必勝著手，如圖 3-21。

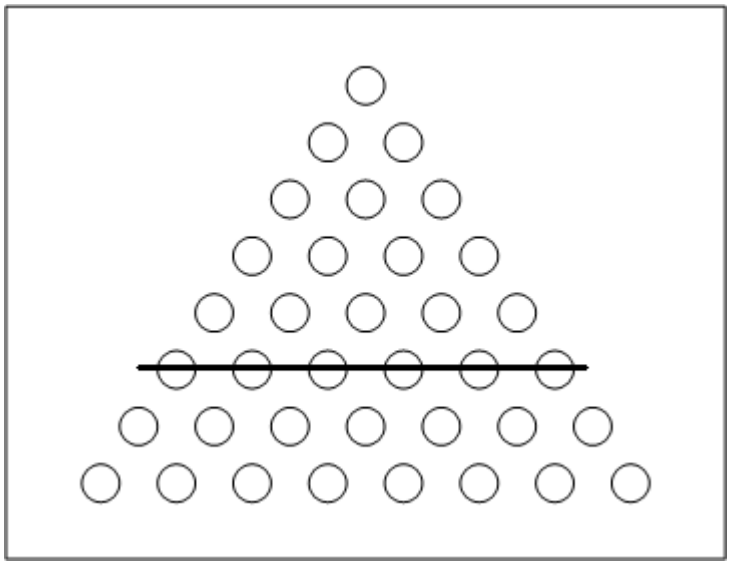


圖 3-21 規則為下到最後一子為敗，八層三角殺棋必勝的第一步

第三節　壓縮盤面狀態資料結構

對三角殺棋切一刀將三角殺棋分割為兩個子盤面的策略在第五種到第七種測試時就須動用到 8G Bytes 的記憶體，若不希望動用到系統的虛擬記憶體，則必須改變策略，犧牲三角殺棋程式搜尋勝負時的速度，在程式中對盤面狀態的資料結構進行壓縮。進而達到節省記憶體的目的，使八層三角殺棋在目前配有 8G Bytes 的個人電腦下，能進行對三角殺棋切一刀將三角殺棋分割為兩個盤面的測試。

對於三角殺棋的盤面狀態的資料結構，每一個狀態勝負其實僅需 1 bit 就可以儲存，其實不需使用要到 1 byte 來儲存。但是以 C 語言來說最小的資料型態為 char，而該資料型態佔用的空間為 1 byte。根據這個簡單的想法，我們即對資料結構做編碼壓縮，使之可以用 1 byte 的空間儲存八個盤面狀態的資訊。

設定盤面狀態陣列資訊的方式，是用兩個匹配陣列達成，如圖 3-22。匹配陣列之所以會設計成這個樣子，目的是為了要和原本的盤面狀態資料陣列做 and、or 運算。因為我們知道在程式語言中，做硬體的 and、or 邏輯運算的速度是很快的，而這樣的設計將可以使八層三角殺棋求勝負解的程式多付出的計算時間降低。

```
unsigned char Compare1[8] ={  {1},  {2},  {4},  {8}, {16}, {32}, {64},{128}};
//將資料陣列設為 1
unsigned char Compare0[8] ={{254},{253},{251},{247},{239},{223},{191},{127}};
//將資料陣列設為 0
```

圖 3-22 壓縮盤面資料的匹配陣列

我們要將匹配陣列設計成如圖 3-22 的原因，則是因為這樣的編碼方式是較為簡單的，我們將兩個陣列的值皆轉換為二進位就可以了解為何會這樣設計匹配陣列，二進位表示法的示意圖如圖 3-23。原始盤面狀態陣列中每個元素原本是用 char 來宣告，該陣列元素所佔大小是 1 byte，也就是 8 bit，可表示的範圍為-128 ~ 127。為了盤面狀態的壓縮，我們將其宣告為 unsigned char，可表示的範圍為 0 ~ 255，這樣的改變使得在程式設計上比較方便。

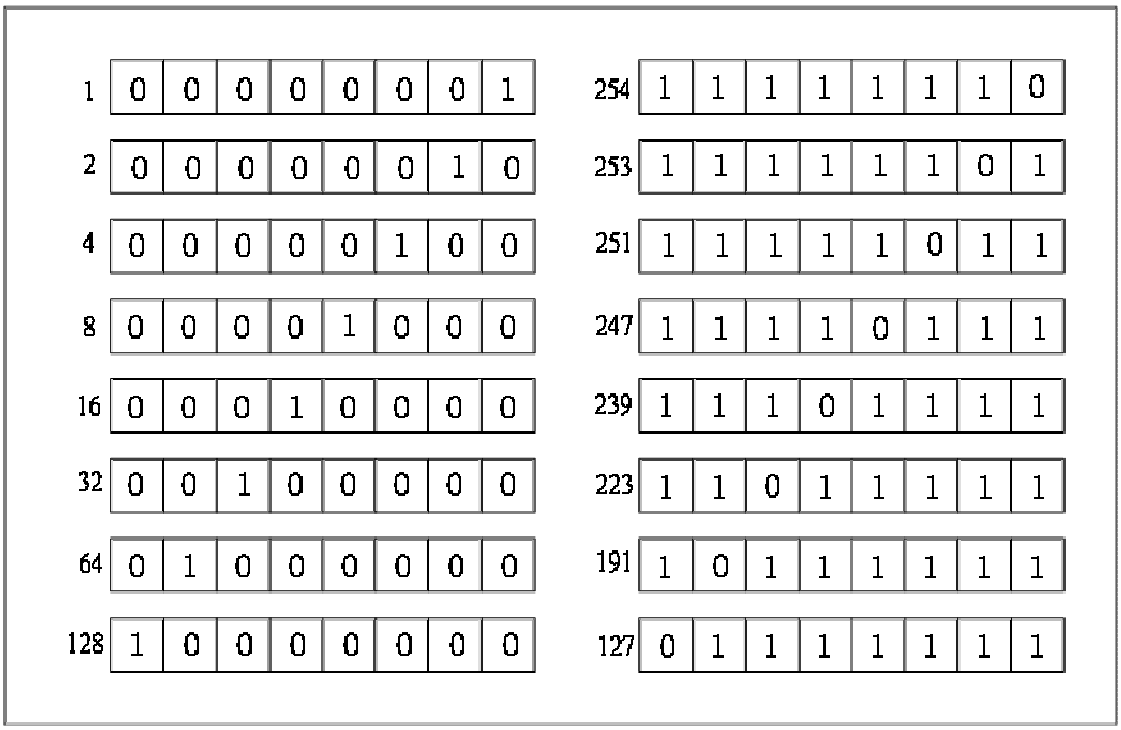


圖 3-23 匹配陣列二進位表示法示意圖

若我們要將盤面狀態陣列的一個元素設為 1，也就是將該盤面狀態設為先手勝，我們只要將該盤面狀態陣列元素與 Compare1 匹配陣列中對應的元素做 or 運算。若我們要將盤面狀態陣列的一個元素設為 0，也就是將該盤面狀態設為先手敗，我們只要將該盤面狀態陣列元素與 Compare0 匹配陣列中對應的元素做 and 運算。

我們將狀態盤面設為 1 之所以要用 or 運算，主要是因為這樣不會影響到其他狀態的值，如圖 3-24。因為其他盤面狀態的值與 0 做 or 運算，其盤面狀態不會改變，只有與 1 做 or 運算會改變。將狀態盤面設為 0 用 and 運算也是同樣的道理。其它盤面資料與 1 做 and 運算，其結果將不會改變原本的值，盤面資料原本不論是 0 或 1，對其與 1 做 and 運算後，都還是原本結果。只有與 0 做 and 運算才會改變結果，如圖 3-25。

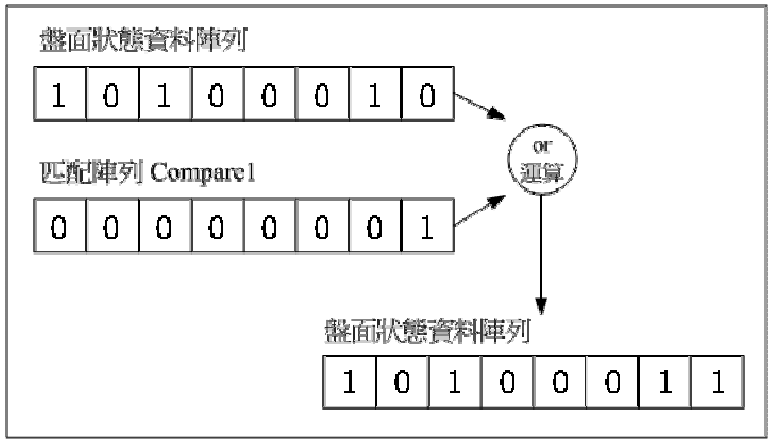


圖 3-24 與匹配陣列作 or 運算

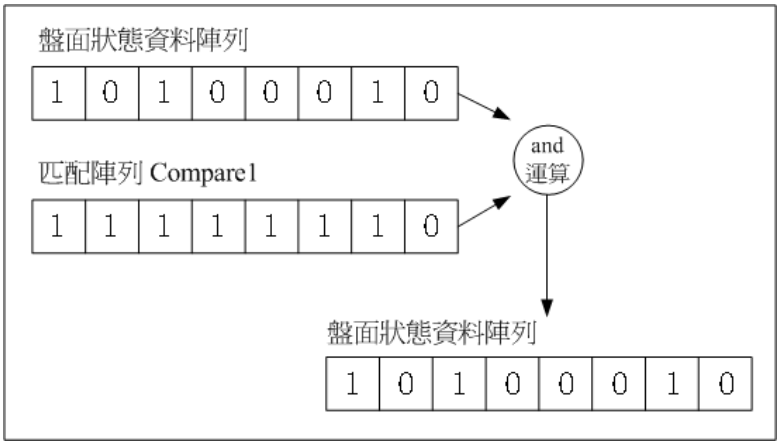


圖 3-25 與匹配陣列作 and 運算

要尋找對應的 Compare0 匹配陣列元素或 Compare0 匹配陣列元素時，我們是使用簡單的求餘數計算。舉例來說，若以八層三角殺棋以倒推法從最後一個盤面往前倒推時，遇到的第一個盤面狀態為 2^{36}-2，也就是 68719476734，但因為現在資料壓縮的關係其真正儲存的盤面狀態陣列的索引值必須先除以 8，以便取得真正的盤面狀態陣列的索引值。而儲存在盤面狀態陣列裡的位置是由除以 8 之後的餘數來決定，其餘數的值會介於 0~7 之間，我們就利用餘數的值來尋找對應的匹配陣列，這麼一來就可以達到資料壓縮的目的。資料壓縮演算法的簡單示意圖如

圖 3-26。

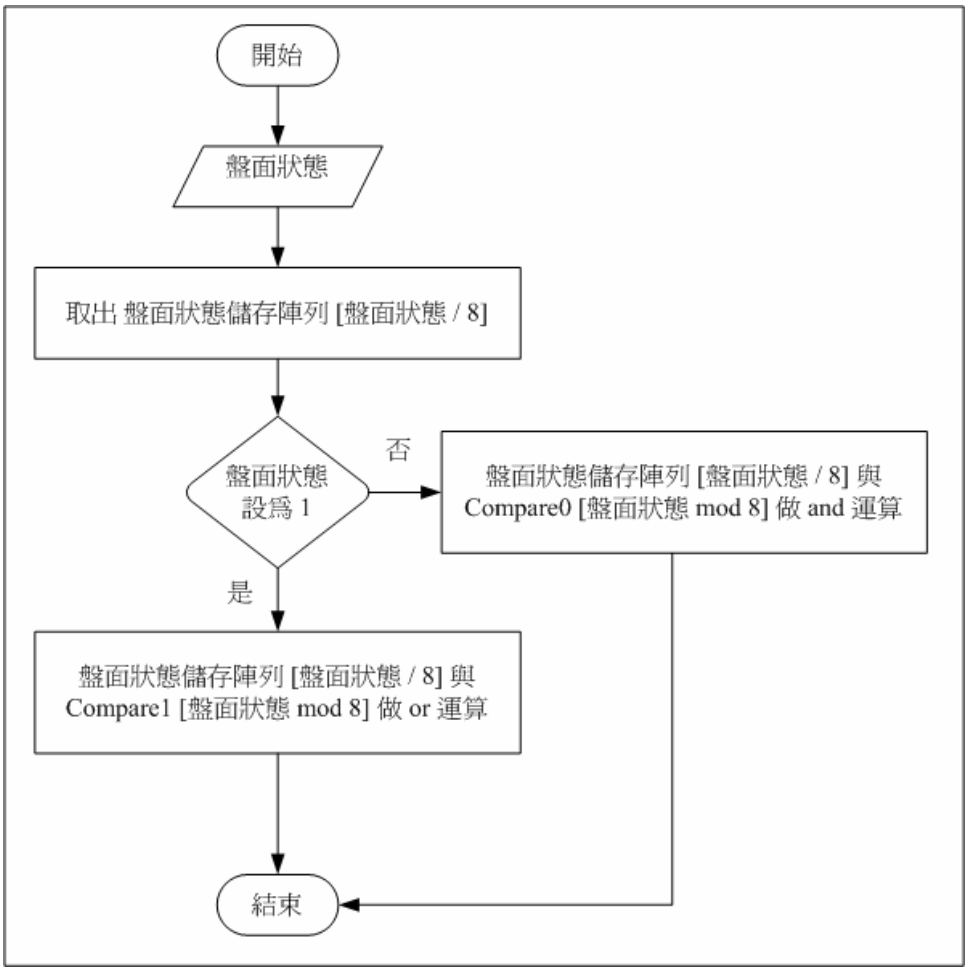


圖 3-26 壓縮盤面狀態之資料處理示意圖

第四節　改進倒推法的記憶體管理

在第三節中，我們觀察三角殺棋三到七層的必勝著手，進而推導到八層三角殺棋盤面中第一手可能會下在何處的這個方法，很可惜僅有找到三角殺棋規則為取得最後一子為敗的結果。所以三角殺棋規則取得最後一子為勝就只能繼續沿用許舜欽教授的倒推法，利用該方法窮舉所有可能盤面進而求出八層三角殺棋的結果。

我們程式的開發是在 CPU 為 AMD Athlon64 X2 4000+ 2.1GHz、記憶體為 8G Bytes、作業系統為 Microsoft Vista 64 位元的版本、編譯器為 Microsoft Visual Studio 2007 的一台個人電腦上。在該環境之下程式雖然能充份使用 8G Bytes 的記憶體，但卻有著每個陣列最多宣告 2G Bytes 的限制。因為編譯器有這樣的限制，故我們將 8G Bytes 的儲存空間切割為四個 2G Bytes 空間。若假設一開始是在 Unix 環境之下做研究，則就沒有每個陣列 2G Bytes 的限制，在撰寫程式上會比較方便，但就不會發現到這樣分割的方式。

若要將八層三角殺棋的每個盤面資訊皆儲存下來，每個盤面資訊就算只儲存 1 bit，也必須耗費 8G Bytes 的記憶體。再加上作業系統本身所耗費的記憶體，則必會動用到虛擬記憶體，而虛擬記憶體相對於實體記憶體的速度是極慢的。若我們一次向系統要求 8G Bytes，則勢必會影響到八層三角殺棋求解的時間。

為了降低這個問題所造成的影響，我們使用一個記體體管理策略，程式有需要該記憶體時才配置給它。前面我們已經將八層三角殺棋所需的 8G Bytes 盤面資訊分割成四段 2G Bytes 的記憶體，如圖 3-27。我們即可以在程式求解的過程中依程式的需求，根據記憶體的需求再向系統要求記憶體。如此一來在八層三角殺棋求勝負解程式執行的過程中，該程式的記憶體需求會隨著程式一步步求解的需要，依序向系統要求 2G Bytes，共四次。

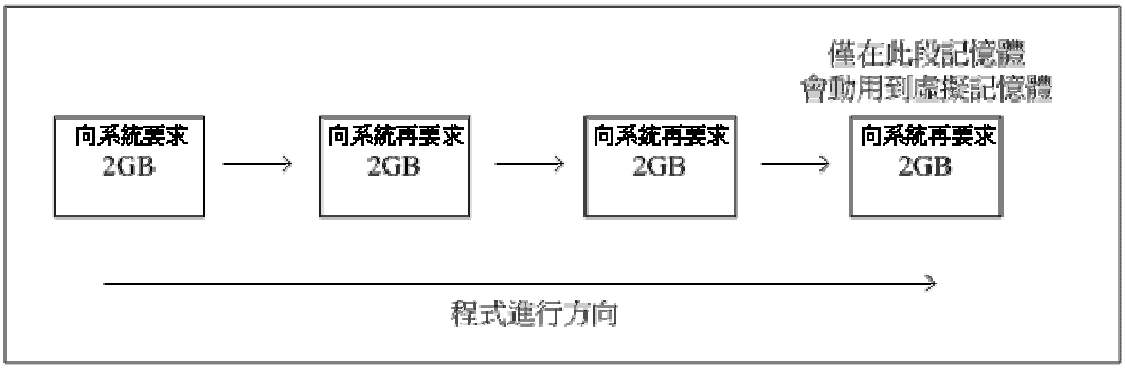


圖 3-27 記憶體要求示意圖

對三角殺棋盤面資訊記憶體分割的主要概念雖然簡單，但是在設計程式的層面則必須要小心與謹慎，每個區塊間如何與其他區塊的記憶體作存取，將是設計三角殺棋求解程式中的主要關鍵。由圖 3-28，我們可以清楚的知道每段記憶體向其他記憶體區塊要求的情況。

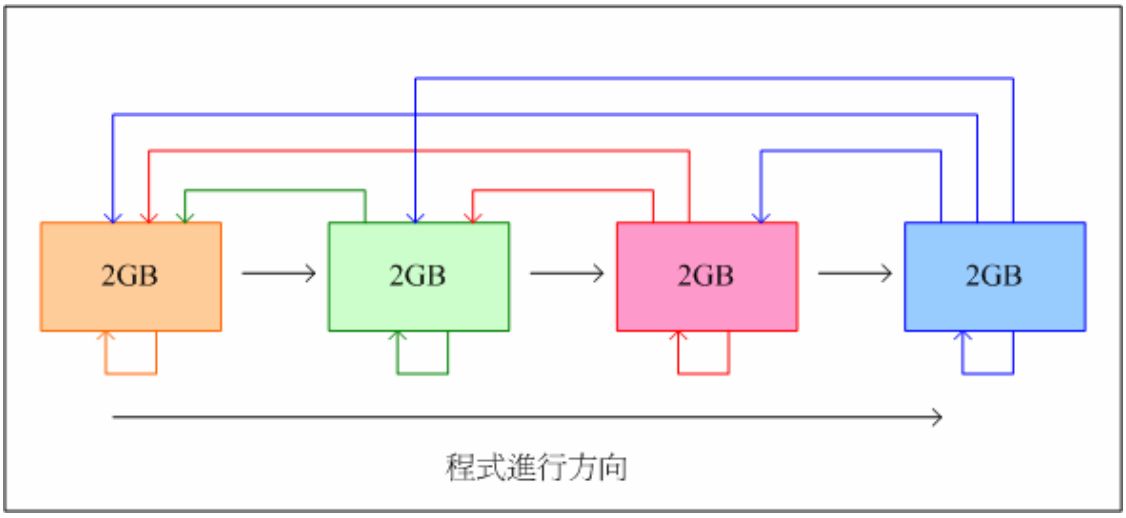


圖 3-28 記憶體區塊關係

記憶體區塊之所有會有這樣的需求關係，主要是因為觀察倒推法中存取盤面狀態資料的程式碼，會影響到存取盤面狀態的程式碼主要是目前盤面與可行著手做 or 運算。所以在設計程式時，我們可以根據每個盤面狀態與可行著手的 or 運

算之後的值，判別計算後的值是屬於哪一個區段的記憶體，再來存取該段記憶體的值。

原本的演算法如圖 3-29，要更改的地方為求出 S(i)下第 j 著手之後的盤面狀態這個地方，必須增加一個處理步驟，就是判斷該盤面狀態是位於哪一個記憶體區塊，判別完畢之後再將該記憶體區塊減去對應的 offset，每個區塊要減去的 offset 如表 3-3。

```
S(2^36-1)= 必勝 ;

for(i=2^36 - 2;i>=0;i--)
{
   S(i) 爲必敗 ;
   for(j=1;j<=288;j++)
   {
       if( 第j著手爲合法 )
       {
           求出S(i) 下第j著手之後的盤面狀態 ;
           if( 下完後的盤面狀態爲必敗 )
           {
               S(i) 爲必勝 ;
               break;
           }
       }
   }
}
```

圖 3-29 原本的演算法

記憶體	區塊 1	區塊 2	區塊 3	區塊 4
offset	6442450944	4294967296	2147483648	0

表 3-3 記憶體區塊 offset 值

所有盤面狀態共有 2^{36}-1 個，也就是 68719476735，盤面狀態分割成四個記憶體區塊之後，第一個記憶體區塊儲存 51539607552~ 68719476735 的狀態，第二

個記憶體區塊儲存 34359738368~ 51539607551 的狀態，第三個記憶體區塊儲存 17179869184~ 34359738367 的狀態，第四個記憶體區塊儲存 0~ 17179869183 的狀態。因為我們有將資料盤面壓縮，所以第一個記憶體區塊減去的 offset 要從 51539607552 壓縮八倍，也就是除以八即為 6442450944，同理，第二個記憶體區塊的 offset 為 4294967296，第三個記憶體區塊的 offset 為 2147483648，第四個記憶體區塊的 offset 為 0，使這些記憶體區塊都能找到對應的盤面狀態陣列。

第五節　倒推法的修改

另外一個改進的方式就是，根據在前面一節中，我們已經推論出八層三角殺棋極有可能為先手勝。也因為在第四次要求 2G bytes 的記憶體時才會動用到虛擬記憶體，故我們將程式的結束條件變更為「若找到第一手可行著手會造成三角殺棋先手勝」。如此一來，我們就可以再替八層三角殺棋求解程式減少一些時間。

而要如何判斷何者為第一手可行著手？在程式撰寫上也不會太困難。若一個盤面資訊恰等於可行著手，即代表此盤面若下該可行著手為第一手，則此盤面為先手勝或先手負。而此盤面資訊若為負，則表示八層三角殺棋為先手勝，但若為勝則沒有任何結論。倘若八層三角殺棋求解程式未找到任何第一手的可行著手，則代表八層三角殺棋為先手負。

經由改良後的演算法我們順利找到了八層三角殺棋的結果，其必勝的第一步可行著手的結果如圖 3-30。

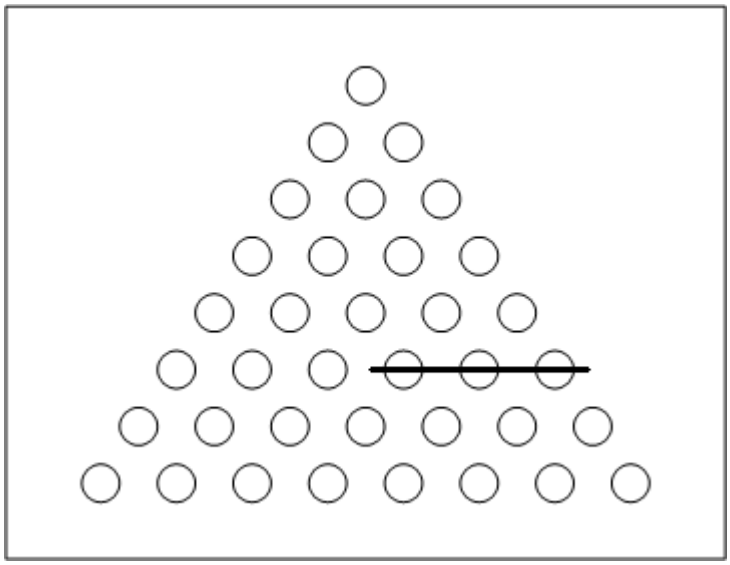


圖 3-30 規則為下到最後一子為勝，八層三角殺棋必勝的第一步

我們的研究過程到此，已經證明了八層三角殺棋在這兩個規則之下皆為先手勝，是一個對先手有利的遊戲。而我們使用了這個方法共花費了 52431 秒，也就是 14.56 個小時。

第四章　其它改進策略

第一節　尋找所有必勝可行著手

我們證明了八層三角殺棋，不論是哪一種規則皆會是先手勝，也就是八層三角殺棋是個對先手有利的遊戲。由於第三章第五節搜尋第一個必勝走步的方法，花費的時間為 52431 秒，也就是 14.56 小時，還在我們容許的範圍。為了八層三角殺棋的完整性，我們也將兩個規則下的三角殺棋的所有必勝著手都找了出來。

以第三章第四節改進記憶體管理的方法，向系統依序要求 2G bytes 共四次，規則為取到最後一子勝的三角殺棋共花費了 61393 秒，規則取到最後一子敗的三角殺棋共花費了 61452 秒。但若以規則為取到最後一子勝的三角殺棋來說，在不改進倒推法的記憶體管理，直接向系統要求配置 8G byte 的記憶體，則花費了 63027 秒，也就是會多花 63027 - 61393 = 2614 秒。

八層三角殺棋(規則為下到最後一子為敗)搜尋所有的必勝著手結果如圖 4-1（每個線段皆為一個著手），雖然必勝可行著手有六步，但六步著手皆為等價，故只有一個必勝著手，如圖 4-2。

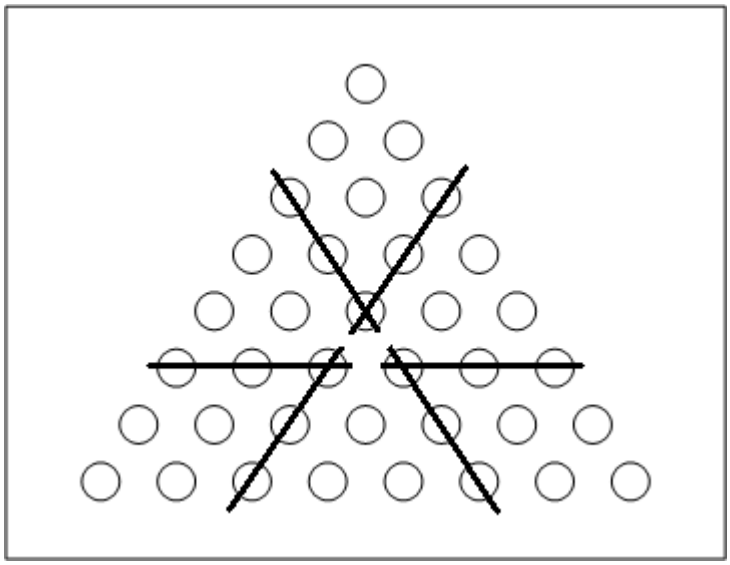


圖 4-1 規則為下到最後一子為勝，八層三角殺棋第一步必勝著手

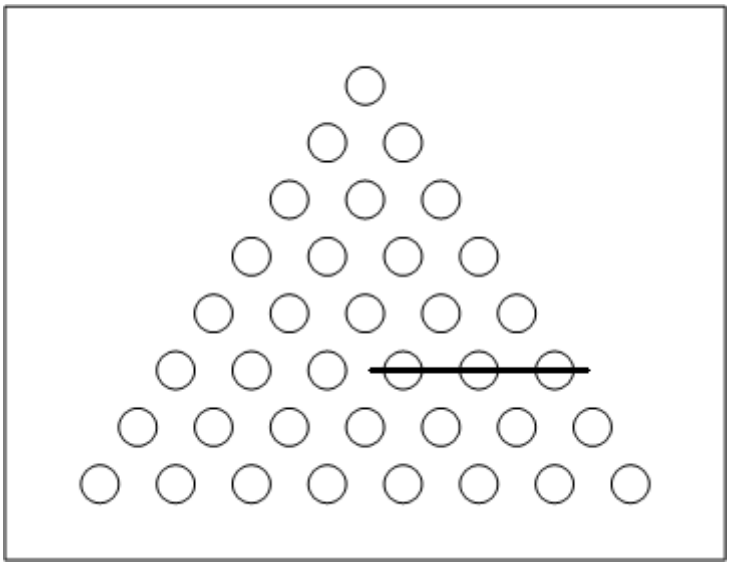


圖 4-2 規則為下到最後一子為勝，八層三角殺棋第一步必勝著手

八層三角殺棋（規則為下到最後一子為敗）搜尋必勝著手結果如圖 4-3（每個線段皆為一個著手），雖然必勝有十二步，但其中九步著手皆為等價，故共有三個必勝著手，如圖 4-4。

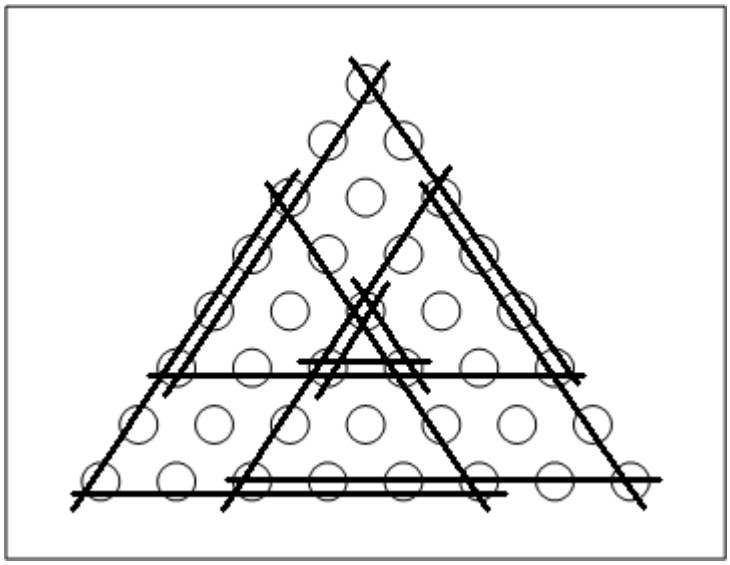


圖 4-3 規則為下到最後一子為敗，八層三角殺棋第一步必勝著手

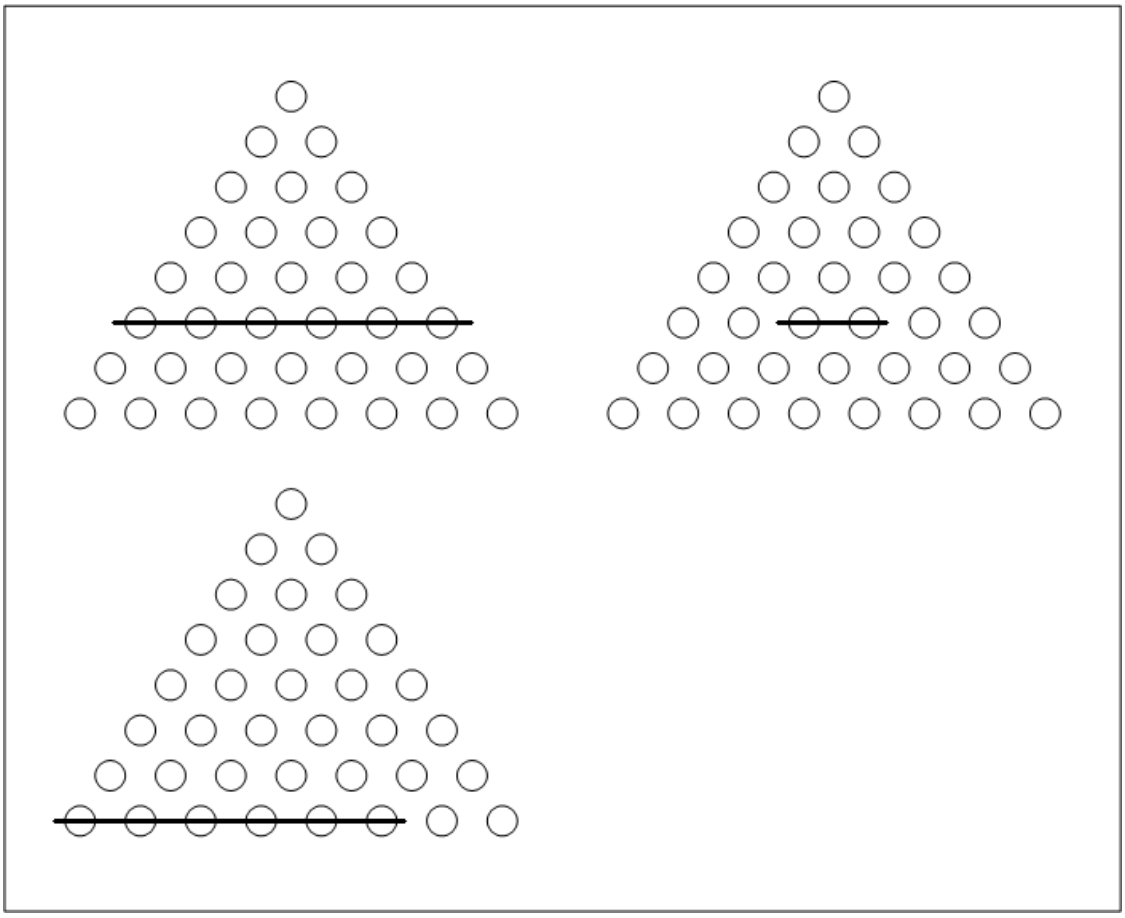


圖 4-4 規則為下到最後一子為敗，八層三角殺棋第一步必勝著手

第二節　刪除不需要的記憶體區塊

我們既然已經知道了每個盤面狀態記憶體區塊之間的關係，就可以針對盤面狀態記憶體區塊做調整。觀察盤面狀態記憶體的特性，將已經不需要的記憶體區塊給刪除。這也是設計這個方法的出發點，但這個方法雖然可以節省記憶體，但在某種程度上會多花費一些時間。

我們已經知道了記憶體關係如圖 4-5，如此一來若要刪除第一個記憶體區塊，那就必須先把第二、第三、第四有用到第一個記憶體區塊的部份先做處理，並且將該部分的值給儲存起來。

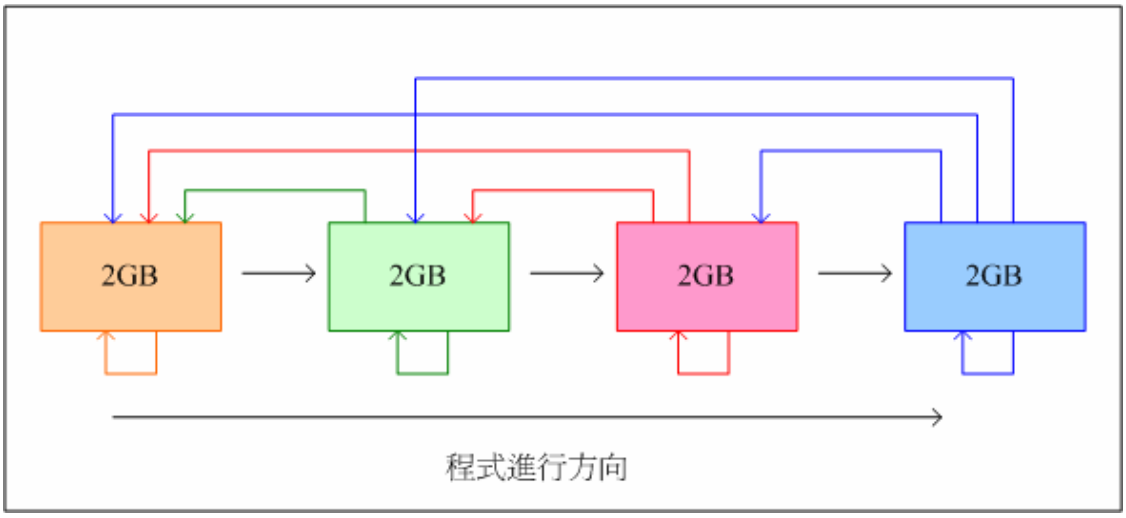


圖 4-5 記憶體區塊關係

我們做了一個實驗，在三角殺棋規則為下到最後一子為勝的情況下，若在處理第四個記憶體區塊時，想把第一個記憶體區塊刪除的話，則必須儲存 2250178498 個對第四個記憶體區塊仍然還有用到的資訊。若推廣到更大的狀態盤面，每刪除一個記憶體區塊，我們還必須對該資訊做整合。

所以，此方法雖然可以用來求多層三角殺棋的勝負，減少了記憶體的需求，但仍需克服一些技術問題，所以構想雖然很好，但實作上會有一些難處，而且花費時間可能會更長。

第三節　三角殺棋盤面編碼的改進

我們歸納了一層到八層三角殺棋的勝負，認為更多層的三角殺棋可能是一個對先手比較有利的遊戲。若我們僅是要知道三角殺棋的勝負結果，我們在程式中可以只尋找一個必勝著手，找到後就立刻結束程式。

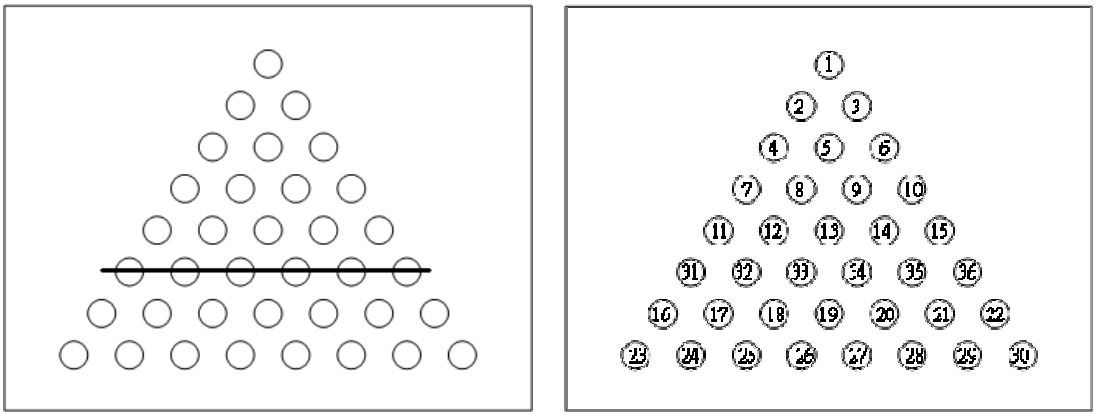
此外，我們若有一個方法，可以有效預測三角殺棋必勝著手，我們也可以將那個可行著手的棋子編號分配較高的權重，然後重新產生對應的三角殺棋編碼。這樣可以加速尋找到那個可行著手的速度，因為我們是用倒推法尋找的，若該可行著手的值越高，則會越快被驗証到。而重新編碼後可行著手產生的演算法如圖4-6。

```
n 值爲可行著手的值，Link 爲連結表

for ( i = 1 ; i <= 36 ; i ++ )
{
        (Link(i,編碼值)-1)
    n = 2                ;
    輸出 n 值 ;
    for ( 方向 = 右邊，左下，右下 )
    {
        n = 2(Link(i,編碼值)-1) ;
        j = i ;
        for ( k = 1 ; k <= 7 ; k ++ )
        {
            if ( Link ( j , 方向) 等於 0 )
            {
                此方向結束 ;
            }
            else
            {
                j = Link ( j , 方向);
                n = n + 2(j-1);
                輸出 n 值 ;
            }
        }
    }
 }
```

圖 4-6 連結表新增編碼值演算法

若要使用圖 4-6 這個演算法，只要在原先連結表格的欄位新增一個編碼值的欄位，如此一來就可以產生對應的可行著手。舉例來說，若我猜測八層三角殺棋的第一步必勝著手為圖 4-7(a)一刀，那我就把位於這一步著手的棋子編號提高，如圖 4-7(b)。如此一來，我們使用倒推法將可以提早搜尋到該著手，這樣就可以節省計算時間。這個方法是比較適合用在有一個好的方法可以猜測必勝的第一步。



(a) 猜測之可行著手　　(b)重新編碼後的三角殺棋

圖 4-7 編碼改進示意圖

第四節　子盤面的重複利用

過去曾經計算過的盤面資料，就某種程度而言其實可以重複再使用。舉例來說，七層三角殺棋的盤面結果相對於八層三角殺棋來說，我們可以把它當作是八層三角殺棋的子盤面。如圖 4-8。

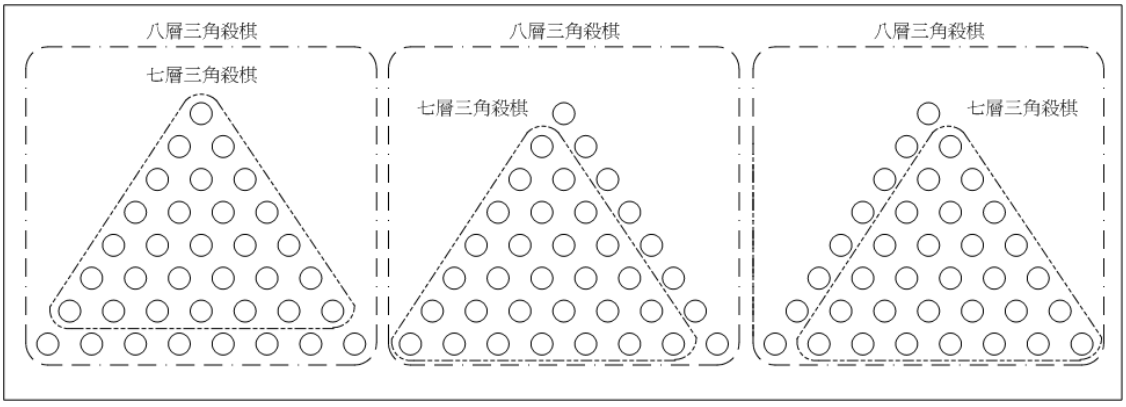


圖 4-8 八層三角殺棋的子盤面

但即使知道七層三角殺棋為八層三角殺棋子盤面的結果，我們也不能直接將全部子盤面的值拿來使用，這樣的盤面狀態如果直接拿來使用，在程式中必須增

加一些判別式。但我們還是可以利用其中之一的子盤面。這麼一來的話在八層三角殺棋求解的程式中，就可以順利使用之前七層三角殺棋求出來的結果，進而達到節省時間的目的。

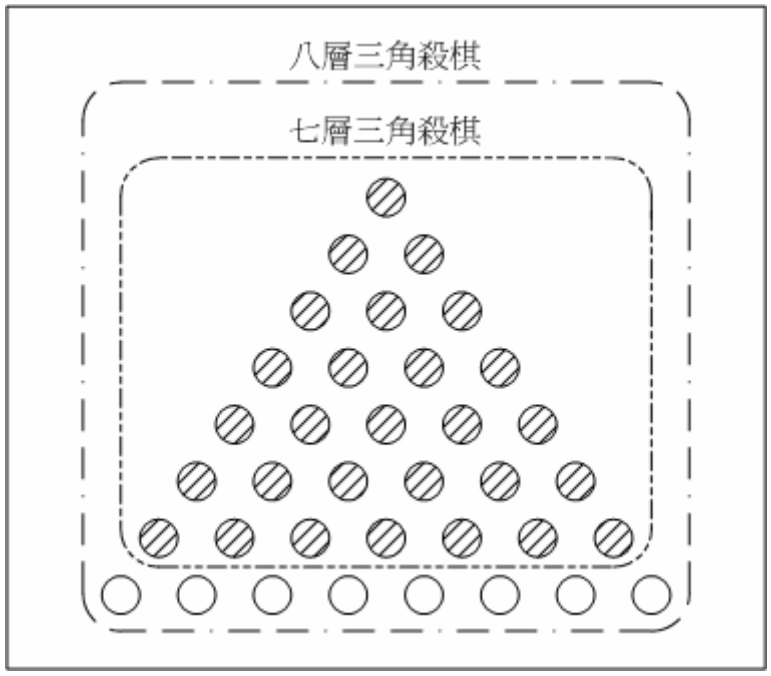


圖 4-9 簡單轉換就可以使用的子盤面

我們可以利用的子盤面如圖 4-9，我們這樣可以將七層三角殺棋的結果放入八層三角殺棋的最後一個記憶體區塊。因為將之看成二進位的值之後可以發現，該子盤面的值介於 11111111000000000000000000000000000 ~ 11111111111111111111111111111111111 ，即 10 進位的 68451041280 ~ 68719476735 之間，又因為倒推法是由最後一個盤面往前倒推，當我們將七層三角殺棋結果放入記憶體後，我們即可直接套用該結果，再繼續由此結果盤面繼續往前倒推出勝負情形。須要注意的是使用此方法還是必須先將七層三角殺棋結果載入至記憶體，不能直接由位置是 68451041280 盤面狀態直接往前倒推，然後把

之後的狀態忽略不計，這樣會造成倒推法的錯誤。

若要套用該結果，則只要將之前的七層三角殺棋子盤面，每個盤面位址加上二進位的 11111111000000000000000000000000000000 ，也就是十進位的 68451041280。盤面狀態位置加上了這個 offset 值之後，即是對應的盤面狀態位置。而為何要加這個值，我們可以參考圖 4-10 就可以很清楚明瞭。

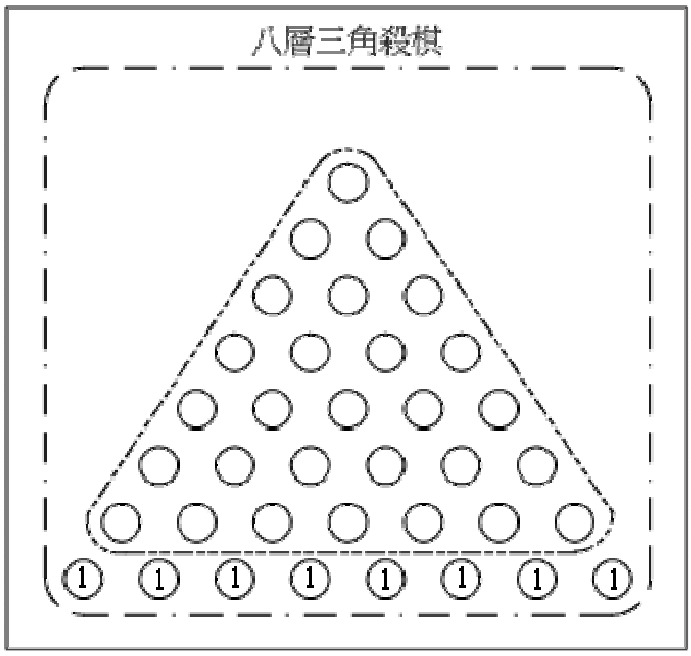


圖 4-10 子盤面修改值示意圖

觀察了圖 4-10，其實就可以瞭解，七層三角殺棋的結果若要可以被八層所使用，只要將原先七層三角殺棋不存在的盤面棋子接補上 1，也就是該棋子被劃殺掉了。如此一來即可以對應到八層的三角殺棋。

第五章　結論與未來研究方向

第一節　結論

本研究證明了八層三角殺棋的兩個規則下的勝負問題，並且將其必勝的可行著手皆尋找了出來。而我們主要的目標是是冀望能夠像 Nim 遊戲一樣，尋找出一個通則來有效解各層的三角殺棋，畢竟三角殺棋是 Nim 遊戲的一種變形。表 5-1 為兩個規則下，一到八層三角殺棋的勝負情形。

規則 / 三角殺棋層數	取得最後一子為敗	取得最後一子為勝
一層	先手敗	先手勝
二層	先手勝	先手敗
三層	先手敗	先手勝
四層	先手勝	先手勝
五層	先手敗	先手勝
六層	先手勝	先手勝
七層	先手勝	先手勝
八層	先手勝	先手勝

表 5-1 一到八層勝負情形

在研究的過程中，我們也針對三角殺棋的特性，提出了多種的改進方法。雖然研究過程中花費許多時間在倒推法上，但我們也研究出來所有先前求出的盤面

是可以運用到往後幾層的三角殺棋。所以在研究的角度上，獲得了許多寶貴的經驗，對往後研究有很大幫助。我們也提出了一個管理記憶體的方式，使得在求解多層三角殺棋的過程中，盤面資訊狀態可以儲存，這樣就可以利用較少量記憶體解多層三角殺棋的結果了。希望這些經驗能作為未來三角殺棋演算法相關研究的參考。

第二節　未來研究方向

我們已經知道各層三角殺棋的結果其實是可以拿來利用的，所以在解九層的三角殺棋時。我們可以把一些比較簡單的子盤面，先利用倒推法求出結果。類似用填表的方式，把各個子盤面儲存在九層三角殺棋的狀態陣列裡。這樣做的話可以節省一些時間。

然而，九層的三角殺棋棋盤面狀態總數成長得更為龐大，以現有想到的方法要想求得解答仍力有未逮，這是值得未來繼續加以研究的一個方向。

附錄 A　程式原始碼

```
// 八層三角殺棋搜尋必勝著手之程式碼
// 本程式碼是將記憶體分割成四段，依序向系統要求 2G Bytes 四次
// 搜尋到一個必勝著手就立即結束程式
// 本程式並且使用記憶體壓縮，1 bit 儲存一個狀態

#include<iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

long long walk[288];   //儲存可行著手數

void readwalk()
{
      int i;
      std::fstream fin;
      fin.open("sort.txt", std::ios::in);
      if (fin == NULL) perror ("Error opening file");

      for (i=0; i<288; i++)
       {
              long long temp;
              fin>>temp;
               walk[i]=temp;
              std::cout <<i<<" - "<<walk[i]<< endl;
       }

       fin.close();
} //  將可行著手數載入到陣列

int soul()
{
      int j; //可行著手
      long long i; //盤面目前狀態
      long long temp; //記錄目前狀態下完某一著手之後所對應到盤面
      long long i8；//記錄程式中目前狀態除上 8 的值，爲了壓縮盤面狀態資料之用
```

```
int m8;   //記錄程式中目前狀態求餘數的值，爲了壓縮盤面狀態資料之用
unsigned char Compare1[8]={{1}, {2}, {4}, {8}, {16}, {32}, {64},{128}};
//將盤面資料狀態設爲 1  的匹配陣列
unsigned char Compare0[8] ={{254},{253},{251},{247},{239},{223},{191},{127}};
//將盤面資料狀態設爲 0  的匹配陣列，本程式不會用到
/*因爲本程式是採用一開始將所有陣列值是爲 0，也就是盤面狀態爲敗。
*/
long long dtemp; //記錄程式中目前狀態下完一著手狀態除上 8 的值，
                 //爲了壓縮盤面狀態資料之用
 int mtemp;      //記錄程式中目前狀態下完一著手狀態求餘數的值，
                 //爲了壓縮盤面狀態資料之用

 int count=287;  //必勝著手倒推

unsigned char *nim4=new unsigned char[2147483648];
 //51539607552~ 68719476735  存在這個陣列
//向系統動態配置 2GB 的記憶體

for ( i =0 ; i< 2147483648 ; i++)
 {
   nim4[i]=0;
 }    //將所有狀態皆設爲敗

 for ( i = 68719476734 ; i >= 51539607552 ; i--)
{
     i8 = (i/8) - 6442450944;
     m8 = i%8;   //之所以要另外設一變數的原因是該變數可能會用很多次
                 //我們不希望重複計算

     for ( j = 0 ; j < 288; j++)
     {

          if   ( (walk[j] & i) == 0)     //這邊判斷是否爲可行著手
          {
               temp =   (walk[j] | i );   //求取目前狀態，下完一著手之後的狀態
               dtemp = (temp/8) - 6442450944; //減掉 offset
               mtemp = temp%8;

               if ( ( nim4[dtemp] & Compare1[mtemp])==0 )
```

```
                {                                                   //這是判斷下著手完後狀態是否爲敗
                    nim4[i8]=( nim4[i8] | Compare1[m8]); //若爲敗，則將目前狀態設爲勝
                    break;                                          //並脫離目前迴圈，停止判斷
                }
            }

        }
        if (i == walk[count])
          {
                printf("i = %lld\n",i);
                if (( nim4[i8] & Compare1[m8])==0) //  若此狀態爲敗
                {
                    printf("found = %lld\n",i);

                    delete [] nim4;
                    return 0;
                }
                count--;
          } //這邊尋找必勝著手

}   //第一段記憶體
 unsigned char *nim3=new unsigned char[2147483648];
//34359738368~ 51539607551  狀態存在這個陣列
//再向系統要求配置 2GB 的記憶體

 for ( i =0 ; i< 2147483648 ; i++)
 {
   nim3[i]=0;
 }

for ( i = 51539607551 ; i >= 34359738368 ; i--)
{
        i8 = (i/8) - 4294967296;    //減掉 offset
        m8 = i%8;
        for ( j = 0 ; j < 288; j++)
        {

            if    ( (walk[j] & i) == 0) //這邊判斷是否爲可行著手
            {
```

```
            temp =    (walk[j] | i );

                if (temp > 51539607551) //若屬於第一段記憶體
                {

                 dtemp = (temp/8) - 6442450944;    //減掉 offset
                 mtemp = temp%8;

                if ( ( nim4[dtemp] & Compare1[mtemp])==0 )
                 {                                          //這是判斷下著手完後狀態是否爲敗
                      nim3[i8]=( nim3[i8] | Compare1[m8]);   //若爲敗，則將目前狀態設爲勝
                      break;
                 }
                }
                else    //若屬於第二段記憶體
                {
                 dtemp = temp/8 - 4294967296;    //減掉 offset
                 mtemp = temp%8;

                 if ( ( nim3[dtemp] & Compare1[mtemp])==0 )
                 {                                      //這是判斷下著手完後狀態是否爲敗
                      nim3[i8]=( nim3[i8] | Compare1[m8]); //若爲敗，則將目前狀態設爲勝
                      break;
                 }
                }
            //nim[i8]=( nim[i8] & Compare0[m8]);
       }
}

if (i == walk[count])
  {
        printf("i = %lld\n",i);
        if (( nim3[i8] & Compare1[m8])==0)
        {
              printf("found = %lld\n",i);
              delete [] nim3;
              return 0;
```

```
                }
                 count--;
        }   //這邊尋找必勝著手
 } //第二段記憶體
 unsigned char *nim2=new unsigned char[2147483648];
 //17179869184~ 34359738367  存在這個陣列
//再向系統要求配置 2GB 的記憶體

for ( i =0 ; i< 2147483648 ; i++)
 {
   nim2[i]=0;
 }

for ( i = 34359738367 ; i >= 17179869184 ; i--)
{
      i8 = i/8 - 2147483648;    //減掉 offset
      m8 = i%8;

      for ( j = 0 ; j < 288; j++)
      {
            if    ( (walk[j] & i) == 0) //這邊判斷是否爲可行著手
            {
                  temp =    (walk[j] | i );

                      if (temp > 51539607551) //若屬於第一段記憶體
                      {

                       dtemp = (temp/8) - 6442450944;    //減掉 offset
                       mtemp = temp%8;

                            if ( ( nim4[dtemp] & Compare1[mtemp])==0 )
                            {                       //這是判斷下著手完後狀態是否爲敗
                            nim2[i8]=( nim2[i8] | Compare1[m8]); //若爲敗，則將目前狀態設爲勝
                            break;
                            }
                      }
                      else if ((temp > 34359738367) && (temp < 51539607552))
                      {                                     //若屬於第二段記憶體
                       dtemp = (temp/8) - 4294967296; //減掉 offset
```

```
                    mtemp = temp%8;


                    if ( ( nim3[dtemp] & Compare1[mtemp])==0 )
                    {                       //這是判斷下著手完後狀態是否爲敗
                        nim2[i8]=( nim2[i8] | Compare1[m8]);
                        break;                          //若爲敗，則將目前狀態設爲勝
                    }
                   }
                   else    //若屬於第三段記憶體
                   {
                    dtemp = (temp/8) - 2147483648; //減掉 offset
                    mtemp = temp%8;

                        if ( ( nim2[dtemp] & Compare1[mtemp])==0 )
                        {                   //這是判斷下著手完後狀態是否爲敗
                        nim2[i8]=( nim2[i8] | Compare1[m8]);   //若爲敗，則將目前狀態設爲勝
                        break;
                       }
                   }

              //nim[i8]=( nim[i8] & Compare0[m8]);
          }
     }
     if (i == walk[count])
       {
            printf("i = %lld\n",i);
            if (( nim2[i8] & Compare1[m8])==0)
            {
                printf("found = %lld\n",i);
                delete [] nim2;
                return 0;
             }
            count--;
       }   //這邊尋找必勝著手
 } //第三段記憶體


unsigned char *nim1=new unsigned char[2147483648];
```

```
 //                0~ 17179869183  存在這個陣列
//再向系統要求配置 2GB 的記憶體
for ( i =0 ; i< 2147483648 ; i++)
 {
   nim1[i]=0;
 }
for ( i = 17179869183 ; i >= 0 ; i--)
{
      i8 = i/8;
      m8 = i%8;
      for ( j = 0 ; j < 288; j++)
      {

           if    ( (walk[j] & i) == 0)    //這邊判斷是否爲可行著手
           {

                temp =    (walk[j] | i );

                    if (temp > 51539607551)    //若屬於第一段記憶體
                    {

                     dtemp = (temp/8) - 6442450944;    //減掉 offset
                     mtemp = temp%8;

                          if ( ( nim4[dtemp] & Compare1[mtemp])==0 )
                          {                              //這是判斷下著手完後狀態是否爲敗
                          nim1[i8]=( nim1[i8] | Compare1[m8]);
                          break;                         //若爲敗，則將目前狀態設爲勝
                          }
                    }
                    else if ((temp > 34359738367) && (temp < 51539607552))
                    {                                                    //若屬於第二段記憶體
                     dtemp = (temp/8) - 4294967296;    //減掉 offset
                     mtemp = temp%8;

                          if ( ( nim3[dtemp] & Compare1[mtemp])==0 )
                          {                          //這是判斷下著手完後狀態是否爲敗
                          nim1[i8]=( nim1[i8] | Compare1[m8]);
                          break;                     //若爲敗，則將目前狀態設爲勝
```

```
                }

            }
            else if ((temp > 17179869183) && (temp < 34359738368))
            {                                                   //若屬於第三段記憶體
             dtemp = (temp/8) - 2147483648;    //減掉 offset
             mtemp = temp%8;

            if ( ( nim2[dtemp] & Compare1[mtemp])==0 )
             {                                  //這是判斷下著手完後狀態是否爲敗
                nim1[i8]=( nim1[i8] | Compare1[m8]);
                                               //若爲敗，則將目前狀態設爲勝
                break;
             }

            }
            else     //若屬於第四段記憶體
            {
                dtemp = (temp/8);
                mtemp = temp%8;

                if ( ( nim1[dtemp] & Compare1[mtemp])==0 )
               {                                  //這是判斷下著手完後狀態是否爲敗
                nim1[i8]=( nim1[i8] | Compare1[m8]);
                                               //若爲敗，則將目前狀態設爲勝
                break;
               }
            }
          //nim[i8]=( nim[i8] & Compare0[m8]);
      }
}

if (i == walk[count])
  {
        printf("i = %lld\n",i);
        if (( nim1[i8] & Compare1[m8])==0)
        {
            printf("found = %lld\n",i);
```

```
                    delete [] nim1;
                    return 0;
                }
                 count--;
            }   //這邊尋找必勝著手
        }
        delete [] nim1;
        delete [] nim2;
        delete [] nim3;
        delete [] nim4;
        return 0;
} //第四段記憶體


int main(int argc, char *argv[])
{
   readwalk();
   soul();
   system("PAUSE");
   return 0;
}
```

附錄 B　八層三角殺棋可行著手編碼表

以下的可行著手集合是根據圖 6-1，也就是八層三角殺棋編碼表，將所有可行著手所代表的值全部計算出來。包含只劃自身一個點、右邊方向、左下方向及右下方向。

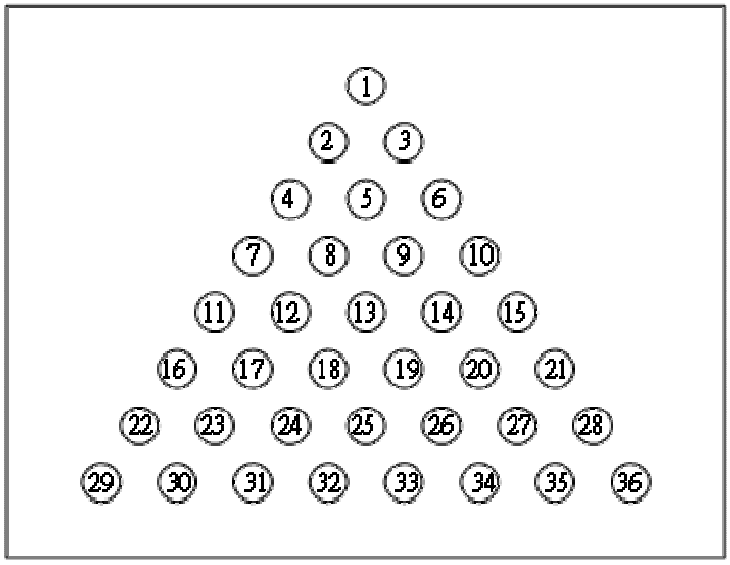


圖 6-1 八層三角殺棋編碼表

	自身一個點	右邊	左下	右下
位置 1	1		3	5
			11	37
			75	549
			1099	16933
			33867	1065509
			2131019	135283237
			270566475	34495021605
位置 2	2	6	10	18
			74	274
			1098	8466
			33866	532754
			2131018	67641618
			270566474	17247510802

位置 3	4		20	36
			148	548
			2196	16932
			67732	1065508
			4262036	135283236
			541132948	34495021604
位置 4	8	24	72	136
		56	1096	4232
			33864	266376
			2131016	33820808
			270566472	8623755400
位置 5	16	48	144	272
			2192	8464
			67728	532752
			4262032	67641616
			541132944	17247510800
位置 6	32		288	544
			4384	16928
			135456	1065504
			8524064	135283232
			1082265888	34495021600
位置 7	64	192	1088	2112
		448	33856	133184
		960	2131008	16910400
			270566464	4311877696
位置 8	128	384	2176	4224
		896	67712	266368
			4262016	33820800
			541132928	8623755392
位置 9	256	768	4352	8448
			135424	532736
			8524032	67641600
			1082265856	17247510784
位置 10	512		8704	16896
			270848	1065472
			17048064	135283200
			2164531712	34495021568

位置 11	1024	3072	33792	66560
		7168	2130944	8455168
		15360	270566400	2155938816
		31744		
位置 12	2048	6144	67584	133120
		14336	4261888	16910336
		30720	541132800	4311877632
位置 13	4096	12288	135168	266240
		28672	8523776	33820672
			1082265600	8623755264
位置 14	8192	24576	270336	532480
			17047552	67641344
			2164531200	17247510528
位置 15	16384		540672	1064960
			34095104	135282688
			4329062400	34495021056
位置 16	32768	98304	2129920	4227072
		229376	270565376	1077968896
		491520		
		1015808		
		2064384		
位置 17	65536	196608	4259840	8454144
		458752	541130752	2155937792
		983040		
		2031616		
位置 18	131072	393216	8519680	16908288
		917504	1082261504	4311875584
		1966080		
位置 19	262144	786432	17039360	33816576
		1835008	2164523008	8623751168
位置 20	524288	1572864	34078720	67633152
			4329046016	17247502336
位置 21	1048576		68157440	135266304
			8658092032	34495004672
位置 22	2097152	6291456	270532608	538968064
		14680064		
		31457280		

		65011712		
		132120576		
		266338304		
位置 23	4194304	12582912	541065216	1077936128
		29360128		
		62914560		
		130023424		
		264241152		
位置 24	8388608	25165824	1082130432	2155872256
		58720256		
		125829120		
		260046848		
位置 25	16777216	50331648	2164260864	4311744512
		117440512		
		251658240		
位置 26	33554432	100663296	4328521728	8623489024
		234881024		
位置 27	67108864	201326592	8657043456	17246978048
位置 28	134217728		17314086912	34493956096
位置 29	268435456	805306368		
		1879048192		
		4026531840		
		8321499136		
		16911433728		
		34091302912		
		68451041280		
位置 30	536870912	1610612736		
		3758096384		
		8053063680		
		16642998272		
		33822867456		
		68182605824		
位置 31	1073741824	3221225472		
		7516192768		
		16106127360		
		33285996544		
		67645734912		

位置 32	2147483648	6442450944		
		15032385536		
		32212254720		
		66571993088		
位置 33	4294967296	12884901888		
		30064771072		
		64424509440		
位置 34	8589934592	25769803776		
		60129542144		
位置 35	17179869184	51539607552		
位置 36	34359738368			

參考著作

[1] Charles L. Bouton，“Nim, A Game with a Complete Mathematical Theory”，The Annals of Mathematics, 2nd Ser., Vol. 3, No. 1/4. (1901 - 1902), pp. 35-39.

[2]“Wikipedia”，網址：http://en.wikipedia.org/wiki/Nim

[3] Alan Tucker，“Applied Combinatorics”，John Wiley & Sons; 3rd edition，June 1994.

[4] 群想網路科技，“ＣＹＣ遊戲大聯盟”，網址；http://cyc165.cycgame.com/cyc/cgi-bin/manual.php?i=manG&game=Nim

[5] 白啓光，“數學嘉年華之數學遊戲” Nim，網址；http://xserve.math.nctu.edu.tw/people/cpai/carnival/game/202.htm

[6] 許舜欽，“三角殺棋的電腦解法及其實現”，電腦季刊，第 16 卷，第 4 期，pp.15-23, Dec. 1982.

[7] 許舜欽，“利用電腦探討七層三角殺棋的勝負問題”，Proc. of 1985 NCS，pp. 798-802，Dec. 1985.

[8] 巫光楨，“尤怪之家之三角棋解析”，網址；http://home.educities.edu.tw/oddest/math222.htm

[9] 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。

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