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

暗棋程式 DarkCraft 的設計與實作

The Design and Implementation

of Dark Chess Program DarkCraft

研究生：施宣丞 撰

中華民國一百零一年六月

摘要

電腦對局在人工智慧領域中佔有很大的份量，它主要分為完全資訊遊戲和不完全資訊遊戲兩大類，完全資訊遊戲是指所有玩家都可以掌握全部資訊的遊戲，例如：圍棋、西洋棋、象棋，不完全資訊則是各方玩家都只能掌握一部分資訊的遊戲，例如：暗棋、麻將、撲克牌遊戲，其中電腦暗棋是近年剛起步，擁有很大發展空間的不完全資訊遊戲。

一個程式的基礎是資料結構，有了好的資料結構才可以讓我們有足夠的基礎去構思出好的演算法，本研究將西洋棋最有效率的資料結構 BitBoard 融合到暗棋裡，讓程式能有很快的執行速度，並改良演算法強化棋力。

本研究已設計並實作了一個暗棋程式 DarkCraft，在 2011 年於荷蘭獲得由國際電腦對局協會(International Computer Games Association)舉辦之電腦奧林匹亞(Computer Olympiad)冠軍。

關鍵字：電腦對局、不完全資訊遊戲、暗棋、位元棋盤

ABSTRACT

Computer games are one of the most important research areas in artificial intelligence. These games fall into two fundamentally different classes: perfect information games and imperfect information games. In perfect information games such as Go and Chess, the current state of the game is fully accessible to both players. In games such as Dark Chess and Poker, the players have imperfect information: they have only partial knowledge about the current state of the game. Particularly, many Dark Chess programs have just been developed in recent years and they have a lot of room to improve their algorithms and implementations.

The data structure is the foundation of programming and algorithm design. A proper data structure can improve code efficiency. In this research, we apply the BitBoard data structure which was commonly used in chess programs to build our Dark Chess program DarkCraft. We also investigate the special characteristics of Dark Chess to fit the requirements of BitBoard and hence reduce the computation time. Furthermore, some techniques are integrated to improve the playing strength.

In this study, we have designed and implemented a Dark Chess program DarkCraft which participated in the 16th Computer Olympiad held in Tilburg, the Netherlands in 2011, and won the gold medal in the Dark Chess tournament.

Keyword: Computer Games, Imperfect Information Games, Dark Chess, BitBoard

致謝

感謝指導教授林順喜博士，指導本論文之研究內容與寫作方向，讓我有機會到歐洲比賽，增廣見聞、為校爭光。

感謝研究室的學長莊台寶學長、黃士傑學長、魏仲良學長、劉雲青學長、白聖群學長、謝政孝學長、李明臻學長、勞永祥學長、陳志宏學長、蔡忠賢學長、唐心皓學姊、學弟妹張懷文、詹凱翔、林澤沅、陳倉裕、陳新颺

特別感謝陳志宏學長經常幫我處理事情。

最後感謝提供我求學至今的父母，使我生活無慮，讓我能順利完成學業，在此僅以本論文獻給我最敬愛的父母。

目錄

摘要 ..... **i**
ABSTRACT ..... **ii**
致謝 ..... **ii**
目錄 ..... **iv**
表目錄 ..... **vi**
圖目錄 ..... **vii**
第一章 緒論 **1**
第一節 暗棋的規則 1
第二節 論文概要 2
第三節 相關論文及程式介紹 3
第四節 近年 Computer Olympiad 比賽紀錄 4
第二章 資料結構相關研究 **5**
第一節 西洋棋 BitBoard 表示法 5
第二節 西洋棋 BitBoard 產生小兵走步 8
第三節 西洋棋 BitBoard 產生國王走步 10
第四節 西洋棋 BitBoard 產生騎士走步 11
第三章 搜尋演算法相關研究 **12**
第一節 Game Tree 12
第二節 Min-Max Search 13
第三節 Nega-Max Search 16
第四節 Alpha-Beta Pruning 17
第四章 暗棋與 BitBoard 資料結構 **19**
第一節 簡介 19
第二節 暗棋 BitBoard 資料結構表示法 20
第三節 BitBoard 兵種表示法 21
第四節 BitBoard 行與列的遮罩 23
第五節 遮罩用法 25
第六節 BitBoard 產生一般兵種走步 26
第七節 BitBoard 產生炮之吃子步 29
第五章 BitBoard 與棋盤編號之間的轉換 **34**
第六章 BitBoard 效能測試 **35**
第七章 新的暗棋演算法策略 **37**
第一節 翻子策略 37
第二節 向上傳遞更新法 42
第三節 同分步處理 43

第四節　單向搜尋 45
第五節　寧靜搜尋 46
第八章 結論與未來方向 47
第一節　結論 47
第二節　未來方向 49
第九章 附錄 50
第十章 參考文獻 55

表目錄

表 2.1 西洋棋初始盤面各兵種資訊 6
表 2.2 西洋棋行列遮罩資訊 7
表 4.1 暗棋行列遮罩表 24
表 4.2 一般兵種 32 個位置之走步遮罩表(皆為常數) 27

圖目錄

圖 1.1 2011 年 Computer Olympiad 比賽結果 4
圖 1.2 2010 年 Computer Olympiad 比賽結果 4
圖 2.1 西洋棋範例盤面 5
圖 2.2 黑色小兵所佔的位置 5
圖 2.3 列遮罩範例 rank2 7
圖 2.4 行遮罩範例 fileb 7
圖 2.5 小兵吃子示意圖 8
圖 2.6 位元對照表 8
圖 2.7 國王走步示意圖 10
圖 2.8 騎士走步示意圖 11
圖 3.1 Game Tree 概念圖 12
圖 3.2 Tic Tac Toe 遊戲的 Min-Max Search 範例[6] 14
圖 3.3 圖 3.2 審局分數部分 14
圖 3.4 Alpha-Beta Pruning 示意圖 17
圖 3.5 Alpha-Beta Pruning 較為複雜的範例 18
圖 4.1 32 位元對應位置 20
圖 4.2 BitBoard 範例 20
圖 4.3 範例盤面 22
圖 4.4 piece[15] 22
圖 4.5 red 所有紅子位置的值 22
圖 4.6 occupied 有棋子位置的值 22
圖 4.7 可以用 1248 加法快速算出其代表的十六進位數字 22
圖 4.8 行遮罩的範例 23
圖 4.9 列遮罩的範例 23
圖 4.10 圖 4.3 的盤面和行列做遮罩 25
圖 4.11 從圖 4.9 中找出紅子 25
圖 4.12 炮在第一行時可能有的炮位 30
圖 4.13 圖 4.12 在程式碼 4.2 中消去炮架後可能得到的結果。 30
圖 4.14 處理行之炮位的切割方法 31
圖 6.1 舊版的傳統 MoveGenerate 35
圖 6.2 新版的 Bitboard MoveGenerate 35
圖 6.3 測試盤面一，新版的速度比舊版的快 3.698 秒。 35
圖 6.4 測試盤面二，新版的速度比舊版的快 6.037 秒。 35
圖 7.1 範例盤面 39
圖 7.2 模擬翻出士 39

圖 7.3 模擬翻出炮 39
圖 7.4 32 位元對應位置 39
圖 7.5 比賽獲勝盤面一 41
圖 7.6 比賽獲勝盤面二 41
圖 7.7 同分走步範例盤面 43
圖 7.8 單向搜尋範例盤面 45
圖 7.9 吃不到的例子 45

第一章 緒論

第一節 暗棋的規則

根據維基百科[7]的說明：暗棋，使用中國象棋和一半的棋盤，將棋盤放在格子裡，共有 8x4=32 格，遊戲開始時將所有棋子的背面朝上，讓人看不見棋子是什麼子，亂洗後放到棋盤上，未翻開的棋子稱作「暗棋」或「未翻子」，翻開的稱作「明棋」。雙方輪流行棋，每當輪到自己時可以選擇翻開一顆暗棋、移動自己的子一格或吃對手的子。先行棋的人翻到的子之顏色為己方顏色，對方為另一個顏色。

吃子時須考慮子力順序。除了帥和將不能吃卒和兵，卒和兵可吃帥和將外，較大的棋子可吃同子力和較小子力的棋，較小的不能吃較大的。唯一的特例為：炮需隔一顆棋子才可吃子，而且不用考慮子力大小。當某一方的棋子全部被吃光，另一方就獲勝。以下為子力順序：

帥（將）＞仕（士）＞相（象）＞俥（車）＞傌（馬）＞炮（包）＞兵（卒）

第二節 論文概要

電腦對局一直是人工智慧中很重要的一門領域，主要的目的是讓電腦可以像真人一樣與人類玩家在不同的棋類遊戲裡競爭，不但可以讓真人和電腦練習，也可以讓電腦與電腦競爭。目前有些棋類的人工智慧已經超越人類職業玩家的水準，證明了電腦在棋盤上要超越人類是絕對有可能的事。

琴棋書畫是中國五千年來祖先們所推崇的四門藝術，其中棋藝是最親近大眾的一門藝術。下棋可以提升人們的氣質、素養以及文化水平，還可以修身養性、讓人把心靜下來、提升心靈的層次和專注力，可謂好處多多。若有了高水準的棋類人工智慧，就能讓許多棋手有可以一直練習的對象，使整體人類的棋藝水平大幅提升。電腦對局不論是在學術研究、文化素養、社會水平都有很大的貢獻，十分值得深入研究和推廣。

電腦對局主要分為完全資訊遊戲和不完全資訊遊戲兩種。完全資訊遊戲可以讓雙方玩家完全掌握所有盤面資訊，遊戲本身不含任何機率成分，例如：圍棋、中國象棋、西洋棋；不完全資訊遊戲則是雙方玩家都無法掌握盤面所有訊息，只能利用片段的資訊來搜尋和計算機率來做出判斷。它包含了機率成分，例如：麻將、撲克牌遊戲，而電腦暗棋又是這幾年來最有發展空間的不完全資訊遊戲，並於 2010 年第十五屆電腦奧林匹亞電腦對局競賽中（The 15th Computer Olympiad, 2010），被正式列為競賽項目，目前已有一些論文及程式[1~4]，針對暗棋研究，但都還處於起步階段，尚待更多人投入做更深入的探討。

第三節 相關論文及程式介紹

目前所知，電腦暗棋的首篇論文為「電腦暗棋之設計及實作」[1]，於 2008 年 6 月由國立台灣師範大學資訊工程研究所謝曜安所撰。在這之前都少有人對電腦暗棋程式做深入的研究，這篇論文為首篇深入研究並提出暗棋演算法的論文。此研究所實作出暗棋程式已經可以擊敗當時市面上所有的商業化暗棋程式了。

[1]主要的想法為使用棋盤-棋子映射結構，用一維陣列紀錄棋盤上那些位置有哪些棋子，再記錄每顆棋子分別在那些位置，交叉查詢資料。搜尋部分使用靜態審局函式，並用一般棋類常用的 Nega-Max 搜尋以及 Alpha-Beta 裁減增加程式運算速度，用 Zobrist Hash、循環剪裁避免計算重複的盤面，增加程式效能。

第二篇為「電腦暗棋程式與經驗法則之配合與實作」[2]，於 2008 年 7 月由國立東華大學資訊工程研究所賴學誠所撰，這篇論文主要在討論不完全資訊的遊戲，並使用了大量的經驗法則和遊戲搜尋樹，並使用靈氣系統的概念，靈氣為每顆棋子都會由內向外擴散而造成影響效果，利用這種效果來幫走步打分數，期望可以產生更好的走步。當棋子全部翻開後遊戲就變成完全資訊遊戲了，加深搜尋深度就可以在這時候讓棋力變很強。

第三篇為「暗棋中棋種間食物鏈關係之探討與實作」[3]，於 2010 年 6 月由國立台灣師範大學資訊工程研究所謝政孝所撰，改良先前[1]程式的做法，改善走步生成方式，並使用食物鏈的概念來產生動態審局函數，先前的程式使用靜態審局，不管盤面屬於何種情形，所有棋子的配分權重都不會改變，現在改成會詳細分析目前盤面狀況，將不重要的子扣分、重要的子加分，讓審局變得更精準。謝政孝所開發的程式 Dark Chess Beta，於 2010 年獲得 Computer Olympiad 亞軍。

第四篇為「電腦暗棋之人工智慧改良」[4]，於 2011 年 6 月由國立台灣師範大學資訊工程研究所勞永祥所撰，由於先前的暗棋程式難以在翻子與走子之間選出恰當的決定，殘局也難以解決，正式比賽時常常以和局收場，這篇論文提出新的策略來解決此問題，並提出距離分的概念讓殘局局面得以解決。

第四節　近年 Computer Olympiad 比賽紀錄

圖 1.1 及 1.2 為這兩年 Computer Olympiad 比賽結果

Rank	Program	Country	Title
1	DarkCraft	TWN	Gold medal
2	Dark_chesser	TWN	Silver medal
3	Diablo	TWN	Bronze medal
4	TopGun	TWN	

圖 1.1 2011 年 Computer Olympiad 比賽結果

Rank	Program	Country	Title
1	Modark	TWN	Gold medal
2	Dark Chess Beta	TWN	Silver medal
3	Leave-or-Lose	TWN	Bronze medal
4	Flipper	TWN	
4	Dark_chesser	TWN	
4	Sparrow	TWN	

圖 1.2 2010 年 Computer Olympiad 比賽結果

第二章 資料結構相關研究

第一節 西洋棋 BitBoard 表示法

BitBoard 是一種高效率的資料結構[5]，概念是用一串位元來表示棋盤上單一兵種的狀態，用 1 與 0 表示棋盤上指定的位置有無該兵種，西洋棋棋盤剛好是 8×8=64 格，與 64 位元 CPU 相符，目前 BitBoard 已經廣泛使用於西洋棋上。

圖 2.1 西洋棋初始盤面

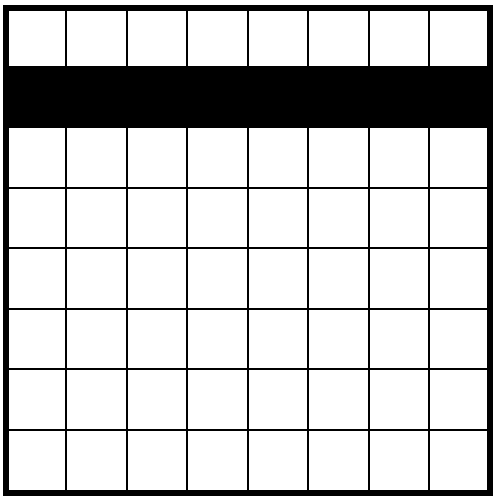


圖 2.2 黑方小兵所佔的位置

圖 2.1 為一個西洋棋初始盤面的棋盤，圖 2.2 為圖 2.1 中黑方的小兵所佔的位置，一共有 64 格，每格與圖 2.1 的對應位置一樣。黑色格子為有黑方小兵的位置，白色格子為沒有，使用一個 64Bits 的無負號整數來儲存圖 2.2 的資訊，黑色格子用 1 代表，白色格子用 0 代表，64 格之中越靠近右上角的部分為越低的位元，在二進位中越靠近右邊。圖 2.2 的內容用二進位來表示的話為：

0000 0000 0000 0000 0000 0000 0000 0000

0000 0000 0000 0000 1111 1111 0000 0000

總共有 64 位數，很長很難看清楚，所以之後一律使用十六進位表示，圖 2.2 的內容用十六進位表示的話為：0x000000000000FF00。

一個變數只能存一種兵種，西洋棋黑白雙方各有六種兵種：國王(King)、皇后(Queen)、城堡(Rook)、主教(Bishop)、騎士(Knight)、小兵(Pawn)，所以需要：K、Q、R、B、N、P 存黑方的子、k、q、r、b、n、p 存白方的子，以下為圖 2.1 中各兵種所存的資訊：

K = 0x0000000000000008	k = 0x0800000000000000
Q = 0x0000000000000010	q = 0x0100000000000000
R = 0x0000000000000081	r = 0x8100000000000000
B = 0x0000000000000024	b = 0x2400000000000000
N = 0x0000000000000042	n = 0x4200000000000000
P = 0x000000000000FF00	p = 0x00FF000000000000

表 2.1 西洋棋初始盤面各兵種資訊

有了這些資訊就可以用邏輯運算很快的產生其他必要資訊，例如想要取得盤面上所有黑子的位置(存入變數black)，只需將黑方的六種兵種做OR的運算：black = K | Q | R | B | N | P 即可，以表 2.1 為例，black = 0x000000000000FFFF，所有白子的位置則為：white = k | q | r | b | n | p，white = 0xFFFF000000000000，棋盤上有子位置為 occupied = black | white，所有空格 empty = ~(black | white)。

棋盤本身也有一些固定的資訊會常使用到，例如棋盤上指定的列或行，以下為兩個範例，這裡的列由下往上數(rank1~rank8)，行由左往右數(filea~fileh)：

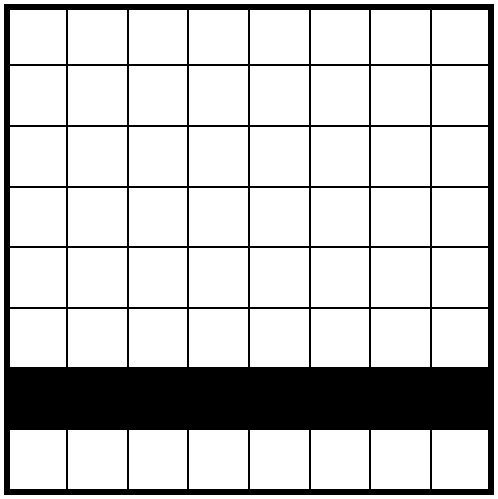


圖 2.3 列遮罩範例 rank2

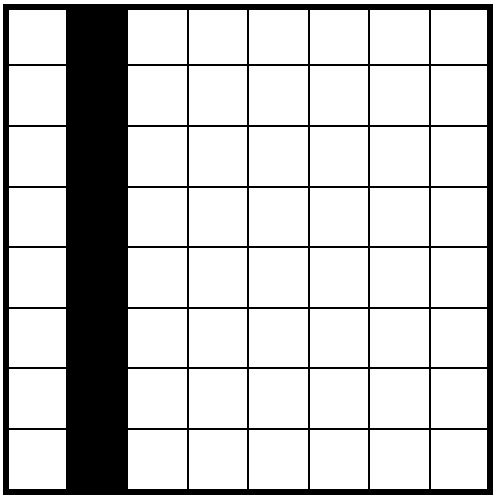


圖 2.4 行遮罩範例 fileb

各列各行的資訊如表 2.2，以下皆為常數：

Rank1 = 0XFF00000000000000	filea = 0x8080808080808080
Rank2 = 0X00FF000000000000	fileb = 0x4040404040404040
Rank3 = 0X0000FF0000000000	filec = 0x2020202020202020
Rank4 = 0X000000FF00000000	filed = 0x1010101010101010
Rank5 = 0X00000000FF000000	filee = 0x0808080808080808
Rank6 = 0X0000000000FF0000	filef = 0x0404040404040404
Rank7 = 0X000000000000FF00	fileg = 0x0202020202020202
Rank8 = 0X00000000000000FF	fileh = 0x0101010101010101

表 2.2 西洋棋行列遮罩資訊

第二節　西洋棋 BitBoard 產生小兵走步

先介紹產生黑方小兵候選著手的方法，小兵第一步可以向前走一格或兩格，之後每次都只能向前走一格，但是若斜前方有敵方的子則可以吃掉對方，這裡先介紹比較重要的吃子部分再介紹走子部分。

當小兵往右斜前方移動吃子時，他的位元會左移七格，使用位元左移運算子「<<」計算，這裡稱呼它為「位元偏移量」。圖 2.5 中左邊的黑色小兵，原本的位置為 0x0000200000000000 若吃右斜前方的子就會變成 0x0000200000000000 << 7 得到 0x0010000000000000，第一次看到這樣的計算可能會覺得很複雜，可以看圖 2.6 的位元對照表，0 代表二進位中最右邊的位數，數字越大越左邊，黑色小兵原本在 45 號位置，向右斜前方移動後變到 52 號位置，數字增加了 7，這樣就能了解為什麼要使用「<< 7」了，能很簡單的推導出，吃左斜前方的子會變成「<< 9」。

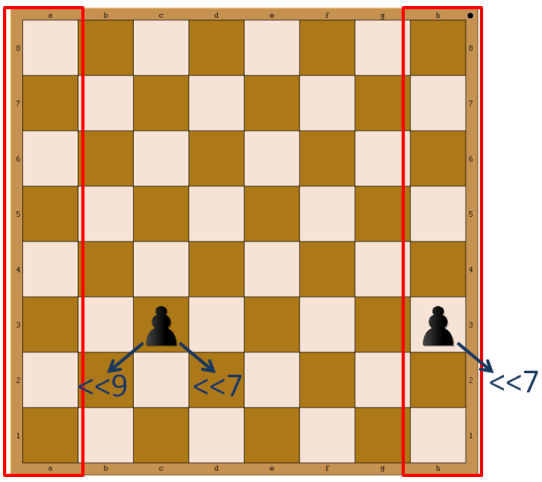


圖 2.5 小兵吃子示意圖

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16
31	30	29	28	27	26	25	24
39	38	37	36	35	34	33	32
47	46	45	44	43	42	41	40
55	54	53	52	51	50	49	48
63	62	61	60	59	58	57	56

圖 2.6 位元對照表

產生小兵吃子步的方法為，若小兵右斜前方或左斜前方的位置是白子即可吃：

PawnCaptures(P) = (P << 7 | P << 9) & white

這裡有一個非常重要的關鍵：若小兵在邊界(圖 2.5 紅色框框部分)，要如何不讓他出界？圖 2.5 右邊的黑色小兵(40 號位置)若使用「<< 7」計算的話就會走到錯誤的位置 47 號，所以上一段的程式碼要修正為，若小兵在最右邊那行(fileh)則不用判斷右前方，左邊也做相同修正，經過精簡修正後的程式碼如下：

PawnCaptures(P) = (((P& ~fileh)<<7 | (P&~filea) <<9)) & white

而小兵走子步的部分，由於小兵只會向前走，走到底時會變成其他棋子，所以不用考慮邊界問題，只要記得小兵第一次可以選擇走一步或兩步就好：

PawnMoves(P) = (P<<8 | (P&rank7)<<16) & empty

要注意的是，黑方和白方的小兵走步方向是不同的，白方需要再另外寫一次。

第三節　西洋棋 BitBoard 產生國王走步

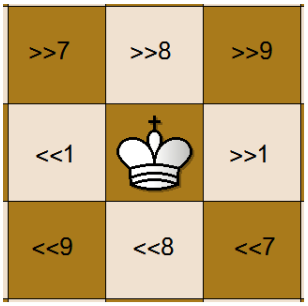


圖 2.7 國王走步示意圖

接著是較為複雜的國王，國王的移動範圍為周圍八個位置，圖 2.7 標示了國王移動範圍裡，每個位置的位元偏移量，以下為範例函式：

```
KingMoves(K)    =
(K & ~rank8) >> 8 | (K & ~rank1) << 8 | (K & ~fileh) >> 1 | (K & ~filea) << 1 |
(K & ~(rank8 | filea)) >> 7 | (K & ~(rank8 | fileh)) >> 9 |
(K & ~(rank1 | fileh)) << 7 | (K & ~(rank1 | filea)) << 9
```

產生國王走步的函式，參數 K 為國王的位置。

先考慮向上一格「>>8」的位置，國王只有在 rank8(最上面一列)時走這步會出界，用(x & ~rank8) >> 8 來避免這個出界問題，國王在 rank8 時就不能向上走了。相對位置為「<<8」的 rank1，再來就是右邊一格的「>>1」在 fileh(最右邊一行)，和相對的「<<1」在 filea。

接下來是對角的四個位置，國王在最上面一列和最左邊一行時，走左上角的「>>7」都會出界，所以當國王在(rank8 | filea)時都不能走「>>7」，將邏輯整理好後就變成(x & ~(rank8 | filea)) >> 7，以此類推可以找出另外三個角落的邏輯，將八個結果 OR 起來就可以產生出國王的所有走步了。

第四節　西洋棋 BitBoard 產生騎士走步

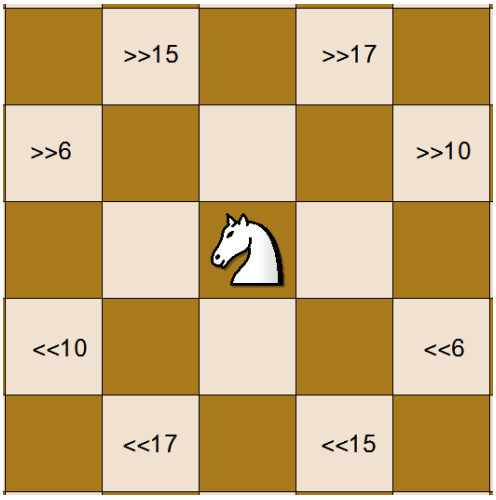


圖 2.8 騎士走步示意圖

騎士走步處理方法跟國王走步很像，如下：

```
KnightMoves(K)    =
(K & ~(rank1 | filegh)) << 6 | (K & ~(rank8 | filegh)) >> 10 |
(K & ~(rank1 | fileab)) << 10 | (K & ~(rank8 | fileab)) >> 6 |
(K & ~(rank12 | fileh)) << 15 | (K & ~(rank12 | filea)) << 17 |
(K & ~(rank78 | fileh)) >> 17 | (K & ~(rank78 | filea)) >> 15 |
```

產生騎士走步的函式，參數 K 為騎士的位置。

filegh = fileg | fileh 為最右邊兩行，其他以此類推。大致上跟國王的差不多，只是騎士橫向或直向可以跳兩格，所以邊界處理變兩層。西洋棋其他子能依以上三種方法推出來。

第三章 搜尋演算法相關研究

第一節 Game Tree

人類下棋時的思考方式為，目前盤面我方先下第一手，對手方用最好的下一手回應，再換我方下第三手，交替下到一定的步數後就會判斷最後的盤面，若很好就下剛剛推演出的第一步棋，若不夠好則從第一手棋開始重新推演。

用樹狀結構畫出的思考過程，稱做對局樹或遊戲樹，每個節點都是一個盤面,根節點是目前盤面狀態,之後的節點都是根節點可能產生的變化,如圖 3.1：

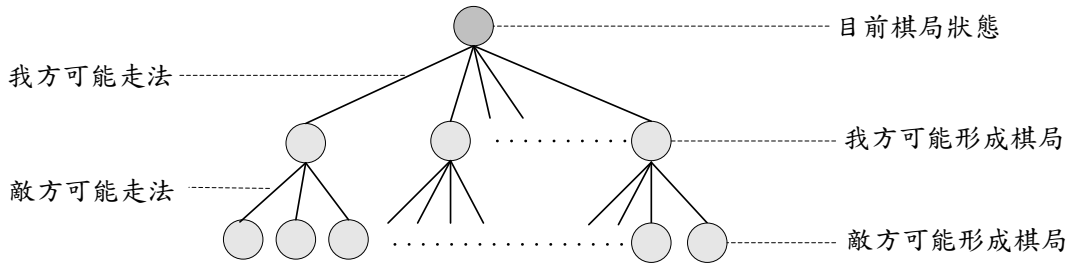


圖 3.1 Game Tree 概念圖

那要如何讓電腦判斷一個盤面好不好呢？我們需要設計一個專門判斷盤面好壞的函式稱為「審局函式」,審局函式的參數為一個遊戲盤面，傳回值為該盤面的分數。簡單的暗棋程式審局方法為：將雙方每顆子都打分數，自己越強的子打越高分，對方越強的子打越低分，審局時將盤面上所有子的分數加起來。

以著名的 Tic Tac Toe(圈圈叉叉)遊戲為例設計一個審局函式:假設我方為 X，當連續三個 X 連成一直線就是我方獲勝，函式會傳回 1 分，若對方三個 O 連成一直線則對方獲勝，傳回−1 分，平手則 0 分。有了審局函式後，下一節就要介紹如何把審局函式和 Game Tree 結合在一起做搜尋。

第二節 Min-Max Search

圖 3.2 的 Game Tree 為一個 Min-Max Search 的範例，每個節點上方的數字為該盤面的審局分數。在剛開始的時候所有盤面是都沒有分數的，分數是搜尋到葉節點(遊戲結束)時才會給予，並向上更新，節點下面的編號為解說用編號。

Game Tree 最上方第零層的根節點為目前盤面狀態，接下來換我方下 X，目前盤面有三個空格，我方有三個位置可以落子，所以 Game Tree 的第一層有三個分支，第一層我方落完子後，換對方落子 O，以 1.1 的分支為例，在 1.1 中還有兩個空格，接下來 O 可以有兩種下法，若 O 下在 1.1.1 的位置則我方落敗遊戲結束，遊戲結束時就需要對該盤面審局，獲得 0 分，若 O 下在 1.1.2 的位置則遊戲還沒結束，換我方落子，在 1.1.2.1 中我方獲勝，遊戲結束我方得 1 分，這時可以知道 1.1.2 的盤面能讓我方得到 1 分，但是在 1.1.1 與 1.1.2 中是換 O 落子，對對方來說，我方分數越低越好，對方一定會選擇對自己有利的下法而選擇-1 分的 1.1.1，所以 1.1 為-1 分，以此類推可以幫整棵 Game Tree 打完分數，最後程式會下第一層分支中最高分的一手 1.3。

Game Tree 每次在我方的層數時，我方一定會選擇最高的分數，但是在對方層數時，要考慮對方也會下最強的一手應對，一定要選擇分支中最低的分數，當整棵 Game Tree 全打完分數時，就會變成一層選擇最高分，一層選擇最低分，所以稱為 Min-Max Search。圖 3.3 為圖 3.2 中審局分數的部分，較好觀察 Min-Max 的狀況。

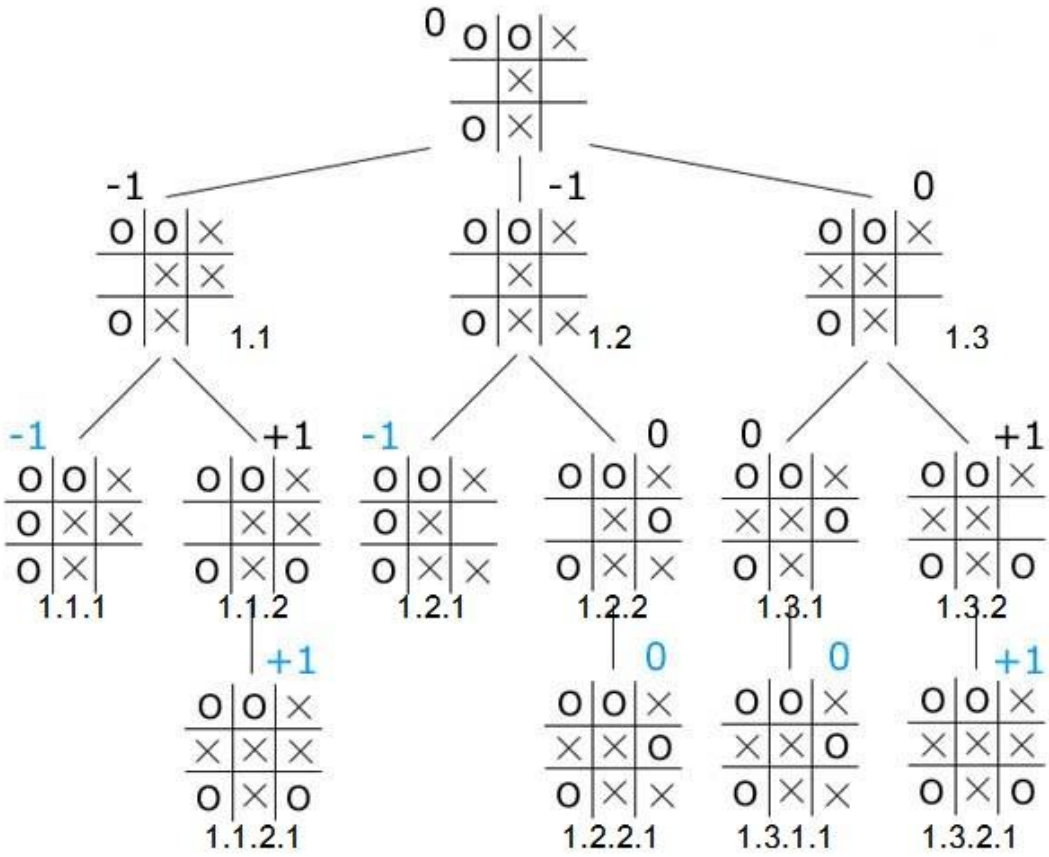


圖 3.2 Tic Tac Toe 遊戲的 Min-Max Search 範例[6]

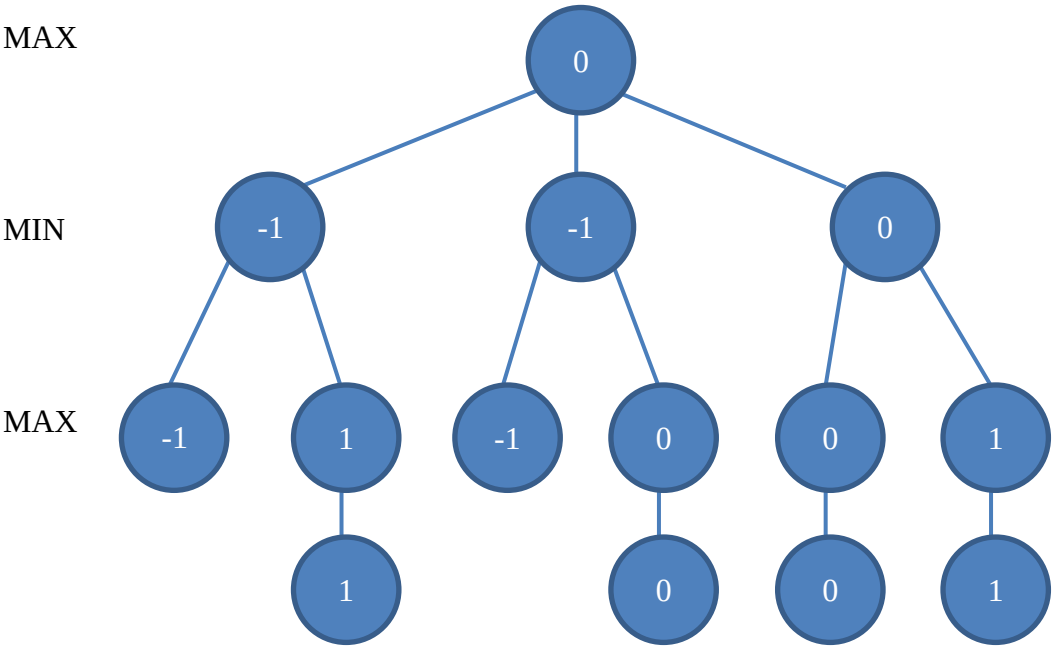


圖 3.3 圖 3.2 審局分數部分

以下是 Min-Max Search 的虛擬碼[2]：

```
int MinMax(int depth){
    int best, value;                //best 儲存最高分用
    if(depth ==0)                   //終止條件
        return Evaluate();          //搜尋到葉節點傳回審局分數
    if(myTurn())                    //如果換我方行棋
        best = -INFINITY;           //我方要取最高分，所以 best 先設負無限。
    else                            //如果換對方行棋
        best = INFINITY;            //對方要取最低分，跟我方相反。
    MoveGen();                      //產生所有合法走步
    while(MoveLeft()){              //搜尋每一個合法步
        NextMove();                 //下一手合法步
        value = MinMax (depth-1);   //遞迴呼叫，層數少一層。
        UnMove();                   //復原走步
        if(myTurn()){               //如果換我方行棋
            if(value > best){       //將最高分記下來
                best = value;
            }
        }
        else{
            if(value < best){       //敵方保留最低分
                best = value;
            }
        }
    }
    return best;
}
```

程式碼 2.1 Min-Max 的虛擬碼

第三節 Nega-Max Search

1975 年 Knuth 和 Moore 提出 Nega-Max Search[5]，來簡化 Min-Max Search 的程式碼，原本的 Search 需要 if(myTurn()) 判斷換誰下，但是分支處理內容的差別只在比大小。只要在遞迴呼叫時加上一個負號，就可以不用分支並得到相同的結果，使程式更簡單易懂，以下是 Nega-Max Search 虛擬碼。

```
int NegaMax(int depth){
    int best, value;
    best = -INFINITY;                    //初始值負無限
    if(depth ==0)
        return Evaluate();

    MoveGen();
    while(MoveLeft()){
        NextMove();
        value = - NegaMax (depth-1);     //這裡多一個負號
        UnMove();
        if(value > best){
            best = value;
        }
    }
    return best;
}
```

程式碼 2.2 NegaMax 的虛擬碼

第四節 Alpha-Beta Pruning

以上兩節兩種 Search 都會在 Game Tree 裡的每個節點都走過一遍，若搜尋 n 層、平均每個節點有 k 種走法，那一共就要拜訪k^n個節點，會花掉許多時間。

1963 年 Alexander Brudno 提出 Alpha-Beta Pruning[5]，他的想法是：搜尋時如果發現接下來不管怎麼做都不會改變對方的最佳分數時，就可以放棄接下來的分支，以節省時間並得到一樣的結果。如圖 3.4[3]，A 會選擇其下方兩個節點中最大的數字，而 B 會選擇其下方兩個節點中最小的數字，現在 B 下方第一個分支是 20，得知 B 的值絕對不會超過 20，所以 A 絕對不會選擇 B，那 B 之後的節點就可以全部不用拜訪了，這個範例可以少拜訪一個節點。另外一個較複雜的範例圖 3.5[3]可以少拜訪 12 個節點。

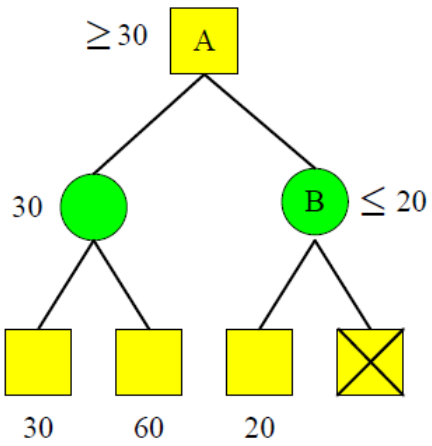


圖 3.4 Alpha-Beta Pruning 示意圖

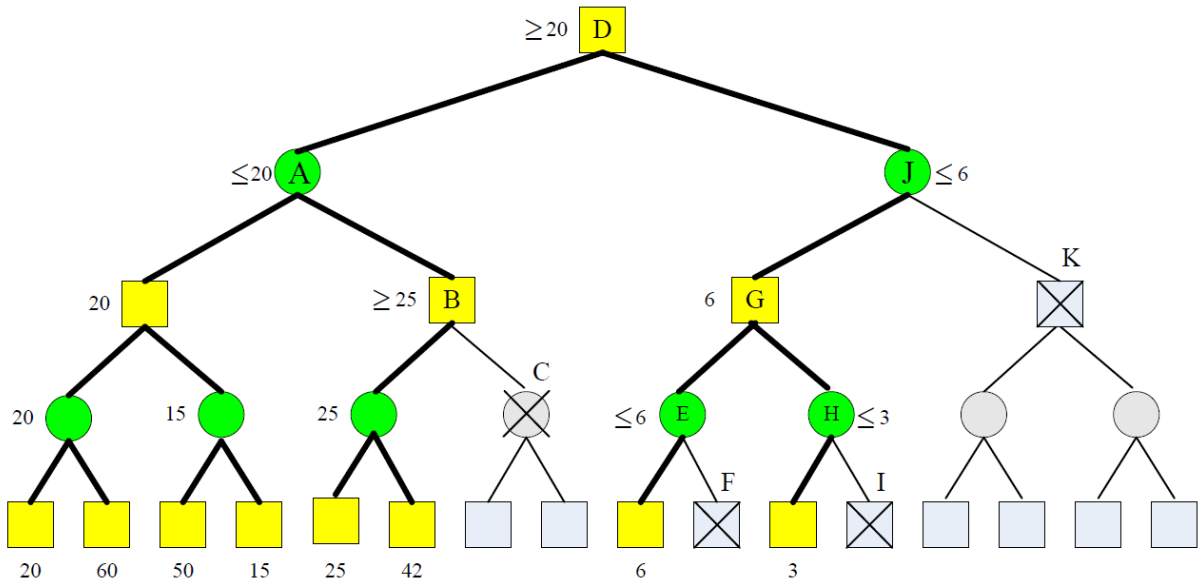


圖 3.5 Alpha-Beta Pruning 較為複雜的範例

```
int AlphaBeta(int depth, int alpha, int beta){
    int value;
    if(depth ==0)
        return Evaluate();

    MoveGen();
    while(MoveLeft()){
        NextMove();             //將 alpah 和 beta 加上負號對調
        value = - AlphaBeta (depth-1, -beta, -alpha);
        UnMove();
        if(value >= beta){      //beta 截斷
            return alpha;
        }
        if(value > alpha){
            alpha = value;
        }
    }
    return alpha;
}
```

程式碼 2.3 NegaMax Search with Alpha-Beta Pruning

第四章 暗棋與 BitBoard 資料結構

第一節 簡介

西洋棋的棋盤一共有 64 個格子，剛好跟 64 位元 CPU 的電腦一樣都是以 64 為一個單位，可以使用第二章介紹的 BitBoard 來儲存資料，並讓程式的執行速度大幅提升。而暗棋的棋盤有 32 個格子，剛剛好跟 32 位元電腦的整數一樣是以 32 為一個單位。但是暗棋之中有一種兵種叫做炮，炮可以跳過多個格子去攻擊對手，所以要將 BitBoard 的技術用在暗棋上是較為困難的，但是我們研究出一種切割與合併的技巧，將暗棋與 BitBoard 做出巧妙的結合。

暗棋與其他棋類遊戲最大的不同是它包含了翻子運氣成份，所以必須計算機率以及期望值來讓好運偏向自己。在電影「決勝 21 點」中，許多麻省理工學院的數學好手們聚集在一起去拉斯維加斯賭場算牌，其中有擅長計算的人、記憶超群的人、也有邏輯出眾的人，組成一個團隊才能在賭場實現 21 點的必勝理論。不過電腦本身就是擁有最強計算以及記憶能力的硬體，這是人類永遠無法超越電腦的地方，我們只需將清晰的邏輯教給電腦，極有可能開發出超越人類棋力很高的暗棋程式。

暗棋最麻煩的地方在於走子以及翻子的選擇，以往其它棋類的搜尋樹只須考慮走子就好，暗棋多了翻子的動作，如何選擇下一步的動作也是一個很有深度的研究問題。另一個難題是這個遊戲很容易和局，如何讓目前優勢的盤面不要走成和局、將劣勢的盤面變和都是很有趣的課題，本研究的意義十分深遠。

第二節 暗棋 BitBoard 資料結構表示法

一個位元可以表示有或無，32 位元電腦的整數資料型態有 32 位元，能表示 32 個位置，剛好與暗棋棋盤有 32 格一樣，我們可以用一個整數來表示一個兵種存不存在於棋盤上指定的位置，使用二進位表示時，存在子的位置用 1 紀錄，不存在的位置用 0 紀錄。為了好觀察，我們以圖 4.1 的直立棋盤表示，31 號代表 2 進位最左邊的位元，數字越大的號碼越左邊，0 號代表最右邊的位元。

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20
27	26	25	24
31	30	29	28

圖 4.1 32 位元對應位置

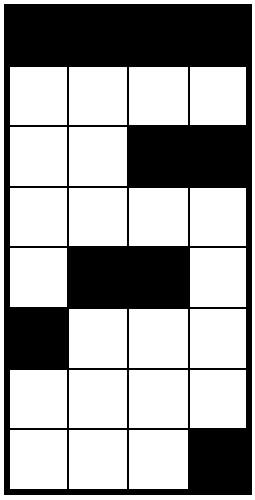


圖 4.2 BitBoard 範例

由於使用位元表示，在編譯器裡編輯程式時，必須使用十六進位表示法才容易觀察變數內容。學過二進位與十六進位的轉換就知道，要將二進位轉換成十六進位最好的方法就是將四個位元為一組計算，將暗棋棋盤直立起來剛好是四行八列，可以很清楚的知道哪四個位元為一組，組成一個十六進位的數字，八列代表有八個十六進位數字。

圖 4.2 為一個範例，棋盤上的黑色格子代表該位置有棋子，白色格子則代表空位，圖 4.2 的二進位值為 0001 0000 1000 0110 0000 0011 0000 1111，十六進位值為 0x1086030F。最上面那列二進位是 1111，就可以知道十六進位中最右邊的值是 F，上面數來第二列是 0000 可以知道十六進位右邊數來第二個值是 0，以此類推可以很快得知 8 個十六進位的值分別是多少，也可以快速從十六進位反推回二進位的值。這裡最重要的就是要注意計算的時候不要將十六進位的左右兩邊用反，棋盤上方的位置靠十六進位的右邊，下方靠左邊。

第三節 BitBoard 兵種表示法

有了 BitBoard 表示法後就可以用它來儲存棋盤上的資料了。我們使用一個無負號的 32 位元整數(unsigned integer，之後內文的整數一律代表無負號整數)，來儲存一種兵種，而暗棋有帥、仕、相、俥、傌、炮、兵、將、士、象、車、馬、包、卒共十四種兵種以及空格、未翻子，需要十六個整數來儲存他們，為了方便撰寫程式，先定義整數型態 U32，再宣告十六個整數跟幾個必要的資訊：

```
typedef unsigned int U32;      // 可以表示 0x00000000 ~ 0xFFFFFFFF
U32 piece[16];      // 各個索引分別代表：0 空格, 1 帥, 2 仕, 3 相, 4 俥, 5 傌, 6 炮, 7 兵, 8 將, 9 士, 10 象, 11 車, 12 馬, 13 包, 14 卒, 15 未翻子。
U32 red, black, occupied;
```

我們必須記錄 14 個兵種、所有空格、未翻子、紅子、黑子、有棋子之格子的位置，有了以上資訊就可以使用邏輯運算快速取得盤面上我們所需要的訊息，若需要其他資訊也可以自行額外加入。

圖 4.3 為一個範例盤面，●為未翻開的子。假設目前盤面是這樣，挑出其中幾組變數(圖 4.4~4.6)來展示各個變數內會存什麼值。圖 4.4 為盤面中 piece[15]未翻子所存的資料，其 16 進位值為 0x90C8081D。

使用圖 4.7 的 1248 加法可以快速計算，例如圖 4.4 中從上數下來第一列為 8+4+1=13=D，則 piece[15]在十六進位中從右邊數來第一個數字就是 D，圖 4.4 中從上數下來第二列為 0+0+0+1=1，就是 piece[15]從右邊數過來第二個數字，以此類推可以算出八個位置的十六進位之值分別為多少。在實作程式時會大量使用這些轉換，熟練後就可以快速從十六進位的值得知盤面狀況如何。

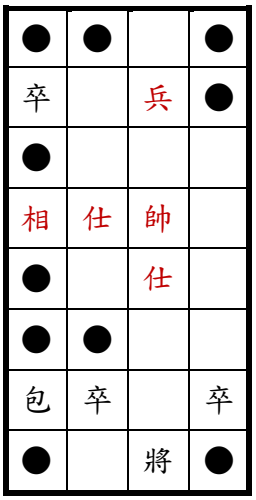


圖 4.3 範例盤面

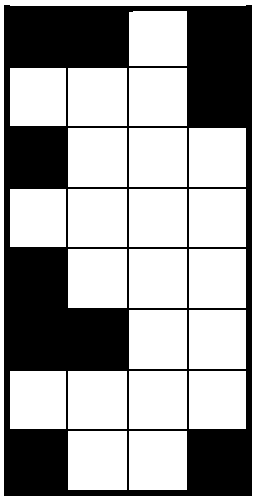


圖 4.4 piece[15]未翻子的值 0x90C8081D

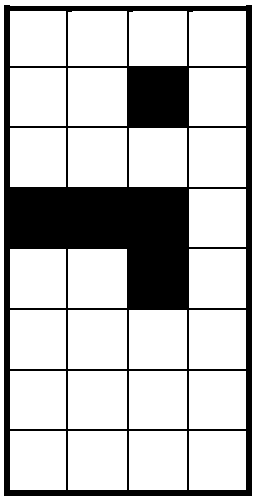


圖 4.5 red 所有紅子位置的值 0x0002E020

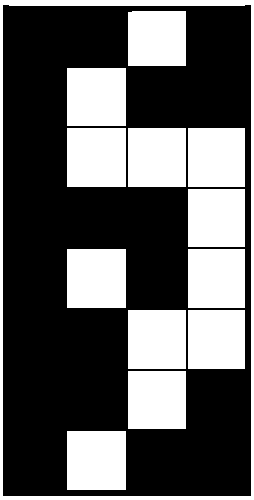


圖 4.6 occupied 有棋子位置的值 0xBDCAE8BD

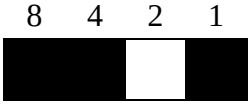


圖 4.7 可以用 1248 加法快速算出其代表的十六進位數字

本範例為 8 + 4 + 1 = 13 = D。

第四節 BitBoard 行與列的遮罩

有了 BitBoard 後還可以利用它做許多事情，我們常常會需要讀取棋盤上指定行或列的資料，這時就能用 BitBoard 行與列的遮罩來作運算。

圖 4.8 為一個行遮罩的範例，一行等於每列都有一個相同的地方被占據一格，例如從右邊數來第二行為 0010 0010 0010 0010 0010 0010 0010 0010，用十六進位表示就是 0x22222222，可以很容易地推出第一行為 0x11111111、第三行為 0x44444444、第四行為 0x88888888，十分好記。

圖 4.9 為一個列的遮罩範例，一列會占據四格連續的位置，例如從上方數來第二列就是 0000 0000 0000 0000 0000 0000 1111 0000，用十六進位表示就是 0x000000F0，能推出第一列是 0x0000000F、第三列是 0x00000F00。

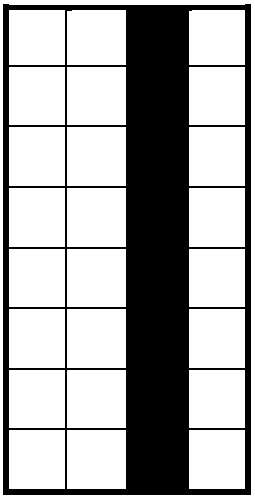


圖 4.8 行遮罩的範例
從右邊數來第二行為 file[1]
0x22222222

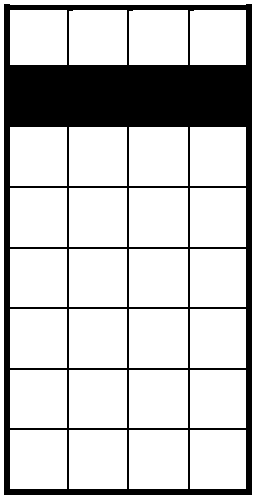


圖 4.9 列遮罩的範例
從上方數來第二列為 rank[1]
0x000000F0

表 4.1 為暗棋行列遮罩查詢表，皆為常數，file 為直的四行，rank 為橫的八列，注意 C++陣列的索引是從 0 開始，所以第一列的索引是 0、第二列是 1，也可以設定 1 號為第一列，但是這樣 0 號的記憶體會浪費掉。

想從上往下數將最上面設為第一列，或從下往上數，沒有一定的規定，可依自己的喜好調整，只要自己覺得方便就可以了，接下來的範例全部以右邊為第一行，上面為第一列。

file[0] = 0x11111111;	rank[0] = 0x0000000F;	rank[4] = 0x000F0000;
file[1] = 0x22222222;	rank[1] = 0x000000F0;	rank[5] = 0x00F00000;
file[2] = 0x44444444;	rank[2] = 0x00000F00;	rank[6] = 0x0F000000;
file[3] = 0x88888888;	rank[3] = 0x0000F000;	rank[7] = 0xF0000000;

表 4.1 暗棋行列遮罩表

第五節 遮罩用法

使用傳統的陣列資料結構，想要取得行列資訊時，都需要去計算哪一行、哪一列在哪裡，但有了 BitBoard 的行列遮罩之後，我們只需要做幾次 AND 的邏輯運算就可以取得我們想要的資訊，以圖 4.3 的盤面為例，若我們想知道跟第四行、第七列的包同行列的哪些位置有子，只需將occupied和第四行(file[3])、第七列(rank[6])各作一次 AND 就可以得到圖 4.10 的結果，若想知道這結果中哪些是紅子的話只需再跟 red 做一次 AND 運算即可得圖 4.11 的結果。

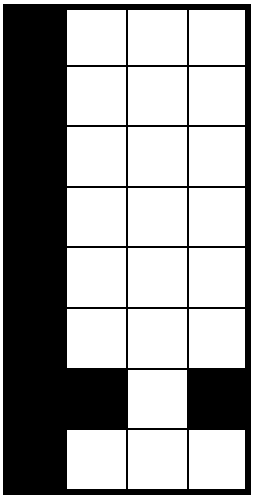


圖 4.10 圖 4.3 的盤面和行列做遮罩

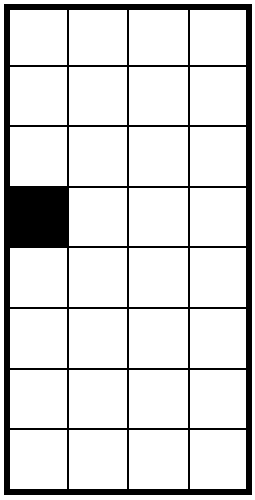


圖 4.11 從圖 4.10 中找出紅子

若想知道其他資訊，可以自行設計新的邏輯，使用 BitBoard 查詢資料都只需幾次 AND 或 OR 邏輯運算就可以得到想要的結果，在大量計算時能比用迴圈計算快上許多。這種資料結構運算十分快速、又容易實作出來，非常有效率。

第六節 BitBoard 產生一般兵種走步

程式在走子之前一定要做產生候選著手的動作，每一手棋的搜尋樹完全展開後可能會有數以百萬的盤面變化，而盤面每變化一次就要再次產生新的著手，資料結構的好壞會大幅影響產生著手的速度，也會直接決定程式的執行速度。

利用 BitBoard 結構產生走步十分簡單又快速，只需要幾個邏輯運算即可，在暗棋遊戲中，炮以外的兵種都只能往隔壁的一格移動和吃子，只有炮在吃子時才可以飛過多個格子，本節先討論產生炮吃子以外的一般兵種之走步方式。

紅子資訊都存在 piece[1]~piece[7]中。黑子為 piece[8]~piece[14]，若要產生紅子之走步只需用迴圈看 piece[1]~piece[7]中之數值是否有不為 0 的值，不為 0 則代表該子有出現在棋盤上，為 0 的話代表棋盤上沒有該子。但一個兵種不一定只會出現在棋盤上的一個位置，例如圖 4.3 的仕所存的值為 0x00024000，仕有兩個，必須先選擇其中一個才可以處理，能用以下 LS1B 函式解決此問題[5]：

```
U32 LS1B(U32 x){    return x & (-x);    }          //Least Significant 1 Bit (LS1B)
```

使用 LS1B( )取得一串位元中最右邊不為零的位元，先將仕的 BitBoard 值丟入 U32 p，再呼叫 LS1B(p)可以得到 0x00004000，取得兩個仕裡面比較上面的那隻仕的位置，經過轉換(第四章講解)得到仕在圖 4.1 中的 14 號位置，將得到的位置存在 U32 src 裡當作起點，現在產生的是仕的吃子步，仕除了將以外的黑子都可以吃，仕的吃子邏輯為「棋盤 14 號位置周圍四格且不是將的黑子」：

```
dest    =    pMoves[14] & ( black ^ piece[8] );      //將得到的位置存到 U32 dest 中
```

pMoves[i]是一般兵種走步的遮罩，為棋盤上某個位置 i 之相鄰位置，例如與圖 4.1 裡的 0 號位置相鄰的有位置 1、4，所以 pMoves[0] = 0x00000012;，走步遮罩皆為常數，如表 4.2，用這種方法只需一行邏輯就可以產生走步，其他兵種只需稍微修改即可，例如帥為的吃子邏輯為「周圍卒以外的黑子都可以吃」:

```
dest   =   pMoves[src] & ( black ^ piece[14] );      //src 為帥在棋盤上的位置
```

而產生走子步的邏輯只需改成「周圍是空點」即可：

```
dest   =   pMoves[src] & piece[0];
```

為了讓大家方便實作，表 4.2 列出 32 個位置的遮罩表。

0	0x00000012	8	0x00001210	16	0x00121000	24	0x12100000
1	0x00000025	9	0x00002520	17	0x00252000	25	0x25200000
2	0x0000004A	10	0x00004A40	18	0x004A4000	26	0x4A400000
3	0x00000084	11	0x00008480	19	0x00848000	27	0x84800000
4	0x00000121	12	0x00012100	20	0x01210000	28	0x21000000
5	0x00000252	13	0x00025200	21	0x02520000	29	0x52000000
6	0x000004A4	14	0x0004A400	22	0x04A40000	30	0xA4000000
7	0x00000848	15	0x00084800	23	0x08480000	31	0x48000000

表 4.2 一般兵種 32 個位置之走步遮罩表 pMove[0]~pMove[31] (皆為常數)

```
for(int i=1; i<8; i++){                 //1~7 為帥~兵，src 為棋子起點，dest 為終點。
    U32 p = piece[i];                   //取得棋子位置
    while(p){                           //將紅色 1~7 號的子都搜尋一遍
        U32 mask = LS1B(p);             //如果該棋子在多個位置，先取低位元的位置。
        p ^= mask;                      //除去位於最低位元的該兵種
        src = GetIndex(mask);           //將最低位元的兵種設為走步起點
        if(i==1)                        //帥，周圍卒(14)以外的黑子都可以吃。
            dest = pMoves[src] & ( black ^ piece[14] );
        else if(i==2)                   //仕，周圍將(8)以外的黑子都可以吃。
            dest = pMoves[src] & ( black ^ piece[8] );
        else if(i==3)                   //相，周圍將、士以外的黑子都可以吃。
            dest = pMoves[src] & ( black ^ piece[8] ^ piece[9] );
        else if(i==4)                   //俥，只能吃車(11)、馬、炮、卒。
            dest = pMoves[src] & (piece[11] | piece[12] | piece[13] | piece[14] );
        else if(i==5)                   //傌，只能吃馬(12)、炮、卒。
            dest = pMoves[src] & (piece[12] | piece[13] | piece[14] );
        else if(i==6)                   //炮，特殊處理。
            dest = CGen(src)     & black;
        else if(i==7)                   //兵，只能吃將(8)、卒(14)。
            dest = pMoves[src] & (piece[8] | piece[14]);
        else
            dest = 0;
        while(dest){                    //如果 dest 有多個位置的話，分開存起來。
            U32 mask2 = LS1B(dest);
            dest ^= mask2;
            U32 result = GetIndex(mask2);
            MT->AddMove(src, result);
        }
    }
}
```

程式碼 4.1 產生紅子吃子步的重點部分

第七節 BitBoard 產生炮之吃子步

由於炮可以跳過多個格子，要用 BitBoard 來處理這個動作十分困難。西洋棋的 BitBoard 中有一個名為 Magic Move-BitBoard Generation 之方法[5]，利用 Hash 和 Bit Shift 來將 BitBoard 做轉換，來產生主教和城堡的走步，但很難實作。本研究在嘗試用此方法產生炮之吃子步的過程中，想出另一種新的切割與合併、且簡單易懂又好實作之技巧，將炮之吃子步和 BitBoard 做出巧妙的結合。

本研究發現 BitBoard 有一種位移特性，表 4.1 中的任一列遮罩都可以經過左移或右移 4 個位元變成另一列，例如第一列左移四個位元變成第二列：

```
0x0000000F   <<   4   =   0x000000F0
```

行也可以，第一行左移一個位元變成第二行：

```
0x11111111   <<   1   =   0x22222222
```

有了這個特性後，就可以利用切割技巧，把跟炮同一行和同一列的資料擷取出來處理。先取得炮在棋盤上 0~31 之位置後存到 src，把 src/4 取得列數(0~7 列)存到 r，src%4 取得行數(0~3 行)存到 c。首先處理跟炮同一列的炮位：

```
U32 x   =   ( (rank[r] & occupied) ^ (1<<src) )   >>   (4*r)
```

將與炮同一列且有棋子的位置用「rank[r] & occupied」擷取出來，為了方便處理，炮本身那格算沒有棋子，用「^ (1<<src)」消去，不管原本在哪一列，全部用「>> (4*r)」移動到第一列統一處理，移動後只有最低的 4 個位元有效(第一列的四個位置)，其他位元都是 0，把結果存入 x 當參數給程式碼 4.2 使用。

這時炮可能會存在於四個位置：從右數來第一、二、三、四行，先處理在第一行的狀況。當炮在第一行時，盤面總共有如圖 4.12 中的八種可能，圖中每一列各為一種狀況，數字為索引，●代表任意一子，○代表炮位(為炮可以吃之對方棋子的位置)，空格代表該位置沒東西。

1				炮
2			●	炮
3		●		炮
4	●			炮
5		○	●	炮
6	○		●	炮
7	○	●		炮
8	●	○	●	炮

圖 4.12 炮在第一行時可能有的炮位

找出○炮位的方法為：將前面得到的參數 x 依照程式碼 4.2 處理，先消去最小位元(炮架)，結果如圖 4.13 所示。若 x 不為零則代表下一個最小位元為炮可以吃子的位置(狀況 5~8)，若為零則代表炮在該列沒有子可吃(狀況 2~4)：

```
if(x){
    U32 mask = LS1B(x);                 //mask 為炮架的遮罩位置，讓 x 消去炮架。
    return (x ^= mask) ? LS1B(x) : 0;   //狀況 5~8 傳回 LS1B(x)，狀況 2~4 傳回 0。
}else   return 0;                       //狀況 1 傳回 0
```

程式碼 4.2 找出炮位的方法

1 ~ 4				
5		○		
6	○			
7	○			
8	●	○		

圖 4.13 圖 4.12 在程式碼 4.2 中，消去炮架後可能得到的結果。

當炮在一列中間的兩個位置就更好處理，若炮在右邊數來第二行只需看他左邊兩個位元是否都為 1，若是則最左邊的位置(8)就是炮位，否則沒有炮位(0)：

```
return    ( (x&12) == 12 )    ?    8    :    0;
```

若炮在右邊數來第三行，將上面的方法反過來處理即可：

```
return    ( (x&3) == 3 )    ?    1    :    0;
```

若炮在右邊數來第四行，將 LS1B(x)改成 MS1B(x)即可(程式如下[5])。

```
U32 MS1B(U32 x){            // Most Significant 1 Bit (LS1B)函式
    x |= x >>    32;        x |= x >>    16;        x |= x >>    8;
    x |= x >>    4;         x |= x >>    2;         x |= x >>    1;
    return    (x >> 1) + 1;  //可以取得一串位元中最左邊不為零的位元
}
```

處理行的吃子步就需要使用切割和合併的技巧，與處理列之吃子步時相同，先用^(1<<src)削去炮並將八個 Bits 全部移動到第一行，如圖 4.14 將該行以炮為中心切成上下兩邊，下半邊使用 CGenCL(U32 x)函式處理，和程式碼 4.2 一模一樣，找出該串位元中第二小的位元即為炮位。上半邊使用 CGenCR(U32 x)，只需將程式碼 4.2 的兩組 LS1B(x)改為 MS1B(x)即可，即為該串位元中第二大的位元，傳回值為圖 4.14 中◯所表示的兩個炮位，將他們用 OR 合併即可。

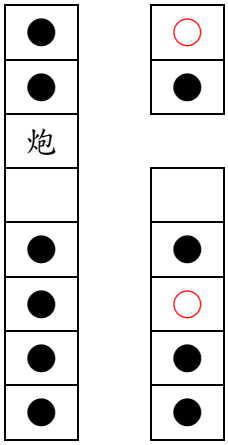


圖 4.14 處理行之炮位的切割方法

接下來介紹切割的方法，炮在計算其一行之炮位時炮可能會存在於八個位置：從上邊數來第一到第八個位置，如圖 4.14，所以需要 CGenC1(U32 x)~CGenC8(U32 x)一共八個函式來處理切割的問題，CGenC1 由於炮在最上邊，執行 CGenCL(x)處理下半邊即可：

```
U32 CGenC1(U32 x){
	return	CGenCL(x);
}
```

炮在第二個位置時，上半邊不可能有炮位，所以可以將上邊兩個位元(含炮本身的位置)去掉再處理下半邊即可：

```
U32 CGenC2(U32 x){
	return	CGenCL(x&0x11111100);
}
```

炮在第三個位置時就需要切成兩邊再合併了，part 為上半邊的炮位，上半邊需要兩個位元都有子時才可能有炮位，且炮位必定在上邊的位元，如圖 4.14，下半邊直接把下邊五個位元取出後執行 CGenCL(x)即可：

```
U32 CGenC3(U32 x){
	U32 part = 0;
	if((x&0x00000011)==0x00000011)
		part |= 1;
	return	part | CGenCL(x&0x11111000);
}
```

第四個位置時上邊切出三個位元，下邊切出四個即可：

```
U32 CGenC4(U32 x){
	U32 part = 0;
	part = CGenCR(x&0x00000111);
	return	part | CGenCL(x&0x11110000);
}
```

第五~八個位置左右對稱，能以此類推簡單找出。

```
U32 CGenC5(U32 x){
	U32 part = 0;
	part = CGenCR(x&0x00001111);
	return	part | CGenCL(x&0x11100000);
}
```

```
U32 CGenC6(U32 x){
	U32 part = 0;
	part = CGenCR(x&0x00011111);
	return	part | CGenCL(x&0x11000000);
}
```

```
U32 CGenC7(U32 x){	return CGenCR(x&0x00111111);	}
```

```
U32 CGenC8(U32 x){	return CGenCR(x);		}
```

最後將一開始移動的位置還原後，把行與列的炮位做一次 OR 運算，再與對手的顏色(red、black)做 AND 運算就可以找出所有的炮位了。

第五章 BitBoard 與棋盤編號之間的轉換

使用 BitBoard 的時候需要常常在 BitBoard 與棋盤編號(整數)之間做轉換，例如圖 4.1 中的 7 號位置，用 BitBoard 表示為 0x00000080，將 7 號轉成 BitBoard 只需將 1<<棋盤編號之位置即可，本範例為 1<<7。

但是 BitBoard 的值直接轉成整數會變成2^7也就是 128，這不是正確答案。我們可以使用 BitHash 的方法，將需要轉換的 BitBoard x 乘上 0x08ED2BE6 再右移 27 個 Bits 即可做出 Perfect Hash 使得 0~31 位置所對應的 BitBoard x 都可以在做完運算後得到另一組 0~31 的對應數字，以下為範例程式碼：

```
index32[32] = {31, 0, 1, 5, 2, 16, 27, 6, 3, 14, 17, 19, 28, 11, 7, 21, 30, 4, 15, 26, 13,
18, 10, 20, 29, 25, 12, 9, 24, 8, 23, 22};
int BitsHash(U32 x){
	return	(x * 0x08ED2BE6) >> 27;
}
int GetIndex(U32 mask){
	return	index32[BitsHash(mask)];
}
```

使用時呼叫 GetIndex(U32 mask)輸入一顆子 x 的遮罩(BitBoard)，會傳回該位置之棋盤編號(整數)，此方法比使用 log 更有效率。

第六章 BitBoard 效能測試

我們所用的測試機器規格如下：CPU Intel i5 M460 2.53Ghz、記憶體 4GB、64 位元 Windows7、32 位元 VC++2008。

測試方法：舊版以傳統的陣列以邊界判斷來產生 MoveGenerate，如圖 6.1。新版以本論文的 BitBoard，如圖 6.2。將測試盤面產生一千萬次 MoveGenerate 來分別計算時間。

```
clock_t start, end;
start = clock();
for(int i=0; i<10000000; i++)
    move_gen(turn, s, d, &n, cap_s, cap_d, &m);
end = clock();
printf("%.6lf", (double)(end - start) / (double)CLOCKS_PER_SEC );

system("PAUSE");
```

圖 6.1 舊版的傳統 MoveGenerate

```
start = clock();
for(int i=0; i<10000000; i++)
    BB->MoveGen();
end = clock();
printf("%.6lf", (double)(end - start) / (double)CLOCKS_PER_SEC );

system("PAUSE");
```

圖 6.2 新版的 Bitboard MoveGenerate

傳統方式執行時間

新版方式執行時間

圖 6.3 測試盤面一，新版的速度比舊版的快 3.698 秒。

傳統方式執行時間

新版方式執行時間

圖 6.4 測試盤面二，新版的速度比舊版的快 6.037 秒。

由圖 6.3 及圖 6.4 可以得知，越複雜的盤面越能看出 BitBoard 的效能。

第七章 新的暗棋演算法策略

第一節 翻子策略

翻子是這遊戲最關鍵的地方，一般人與人對弈時會覺得翻子只是純運氣的動作，但電腦可以把所有可能出現的狀況統計起來，精準計算期望值，這是人類玩家難以做到的事情。目前演算法的翻子時機為：得到盤面資料先做走步搜尋，當目前無法吃到敵方子，且己方沒有子會被吃時就翻子。

暗棋遊戲的目的為：將對方所有子吃光，但不可能一次將對方的子吃光，所以短期目標為:吃掉對方最強的那顆子。而目前暗棋遊戲的殘局非常難處理，所以最好能在中盤時就勝出。綜合以上目標，我們使用的演算法核心策略為：用最快的速度，將對手所有可能被我方吃掉、且子力大的子，以最小的代價吃光，並以此前提翻子，程式碼 7.1 和 7.2 為翻子演算法之虛擬碼和實際程式碼。

```
for(盤面上所有未翻子的位置){
    for(可能翻出之兵種){
        模擬翻出該兵種();
        該位置之分數 += 搜尋(六層);
        結束模擬();
    }
}
選出最高分的位置翻開;
```

程式碼 7.1 翻子演算法虛擬碼

```
void Reveal(int turn){          //參數為換誰下，一方為 0，另一方為 1。
    int maxWeight = -99999999, maxReveal = 0;
    //總共會有幾種子被翻出來(DCountSum)，目前有幾顆未翻子(dCounter)。
    int DCountSum = 0, dCounter = 0;
    U32 dark = piece[15];
    while(dark){                //先計算盤面上有幾顆未翻子
        U32 mask = LS1B(dark);
        dark ^= mask;
        dCounter++;
    }                           //revealWeight 為 32 個位置之翻子分數，初始值為 0。
    if(dCounter >= 31){         //依照個人經驗，第一手翻這四個位置較好。
        revealWeight[9] = 10;
        revealWeight[10] = 10;
        revealWeight[21] = 10;
        revealWeight[22] = 10;
    }
    //模擬翻子前，因為我們翻一手，所以要先將回合交換。
    turn = 1 - turn;            //交換回合
    maxDepth = 5;               //每次把子翻出來後作搜尋的層數，可自行調整。
    int factor = (turn == RED) ? 1 : -1;
    for(int pID=1; pID<15; pID++){
        if(DCount[pID])
            DCountSum += DCount[pID];
    }
    for(int src=0; src<32; src++){      //搜尋盤面上 32 個位置
        if(piece[15] & ( 1 << src )){   //若為未翻子
            for(int pID=1; pID<15; pID++){   //搜尋可能會翻出之子
                if(DCount[pID]){             //若該兵種可能被翻出
                    SimReveal(src, pID);     //模擬該兵種翻出來
                    revealWeight[src] += -( Search(turn) );
                    UnSimReveal(src, pID); //將模擬翻出的子復原
                }
            }
        }
        else{
            revealWeight[src] = -99999999;
        }
    }
```

```
        for(int src=0; src<32; src++){        //找出權重最大的那個位置
            if(maxWeight < revealWeight[src]){
                maxWeight = revealWeight[src];
                maxReveal = src;
            }
        }
        turn = 1 - turn;;            //為了保險起見，將回合換回來。
        bestSrc = maxReveal;    //翻起分數最高的那顆
        }
}
```

程式碼 7.2 翻子演算法實際程式碼

以下為一組簡單的範例，假設目前盤面為圖 7.1 的狀況，尚未翻開的子還有紅炮跟黑士，那麼開始計算時就會先將兵右邊那顆未翻子(圖 7.4 的 4 號位置)做模擬，第一次先將該子變黑士並做搜尋(圖 7.2)，搜尋完後將得到的分數加到 revealWeight[4]，之後將棋盤復原成圖 7.1 的狀態，再將該位置變成紅炮，再做搜尋並將分數加到 revealWeight[4]，4 號位置的模擬就結束了，接下來在 10 號位置做一樣的動作，之後取 revealWeight[4]和 revealWeight[10]較大的值翻開。

卒		兵	●
	●		
相	仕	帥	
		仕	
包	卒		卒
		將	

圖 7.1 範例盤面

卒		兵	士
	●		
相	仕	帥	
		仕	
包	卒		卒
		將	

圖 7.2
模擬翻出士

卒		兵	炮
	●		
相	仕	帥	
		仕	
包	卒		卒
		將	

圖 7.3
模擬翻出炮

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20
27	26	25	24
31	30	29	28

圖 7.4
32 位元對應位置

由於有使用搜尋並計算一定深度內所有可能會發生的狀況，不管盤面如何都能準確算出吃子的期望值，這裡最重要的是配分部份，程式會想盡辦法吃掉分數最高的子,如果那顆子尚未翻出來,程式就會以吃掉那顆子為基準來翻子。

以下為一組範例：假設審局函式的配分為：帥(6000)、仕(2700)、相(900)、炮(2000)，目前為對弈的前期盤面，大部分的子都在，假設帥已翻出，程式在走黑方時會想辦法吃帥，吃帥有兩種吃法：1.用卒、2.用包：

1. 用卒吃：翻帥周圍的子，用卒吃帥的期望值為：能吃掉對方的卒有 5 隻，犧牲的風險有 1 隻將、2 隻士、2 隻象、2 隻包，則獲利估為：5×6000 = 30000，風險估為：6000+5400+1800+4000 = 17200，期望值為 30000-17200 = 12800。
2. 用包吃：獲利估為：6000×2 = 12000，風險估為：0，期望值為 12000-0 = 0。

兩組分數看起來差不多，但全場有五隻卒而包只有兩隻，所以翻帥隔壁的位置比翻包打帥更有機會吃到對方主帥，雖然自己也有被吃的風險，以程式算出來的分數來看，冒險是值得的。當帥被吃掉後，原本第二高分的仕會自動變成最高分的子，翻子時會自動變成被瞄準的對象。以防守的角度來看，程式也會以保護自己大子為前提來翻子。

開始寫程式前，我下棋也是以翻包打帥為主要戰略，但是 DarkCraft 寫出來後程式常翻帥旁邊，原本以為哪邊有 Bug 算錯了，在大量人機測試後，人類玩家的大子經常會在前期被電腦以這種方法吃光，而在正式比賽上也有不錯的表現,獲勝的局面都是在前期就將對方的大子吃光,中盤就贏了,如圖 7.5、7.6：

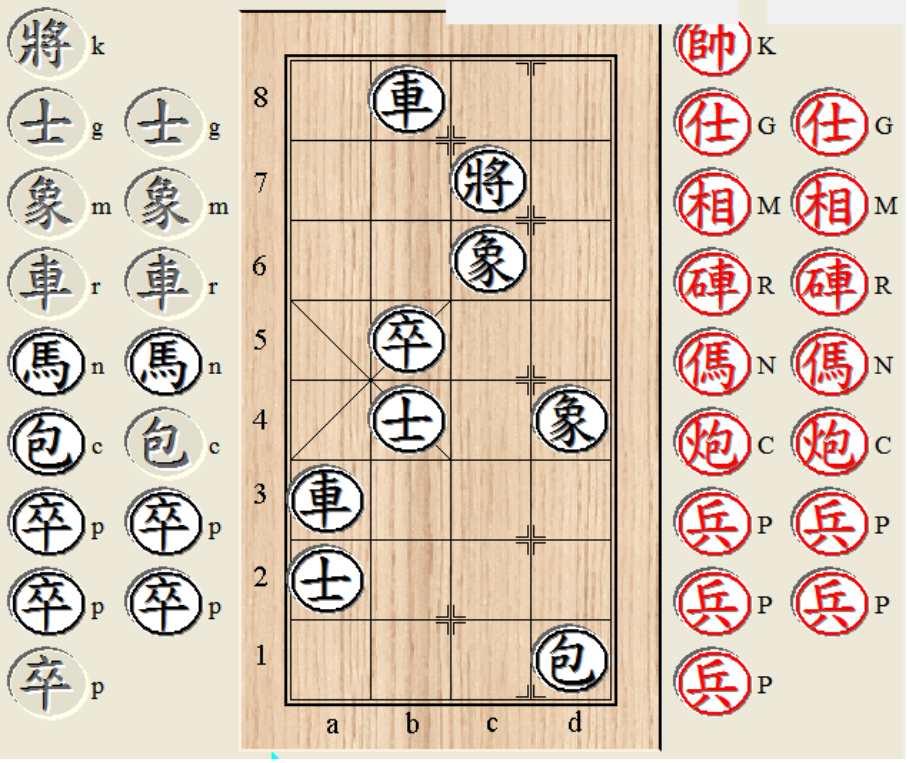


圖 7.5 DarkCraft 比賽獲勝盤面一

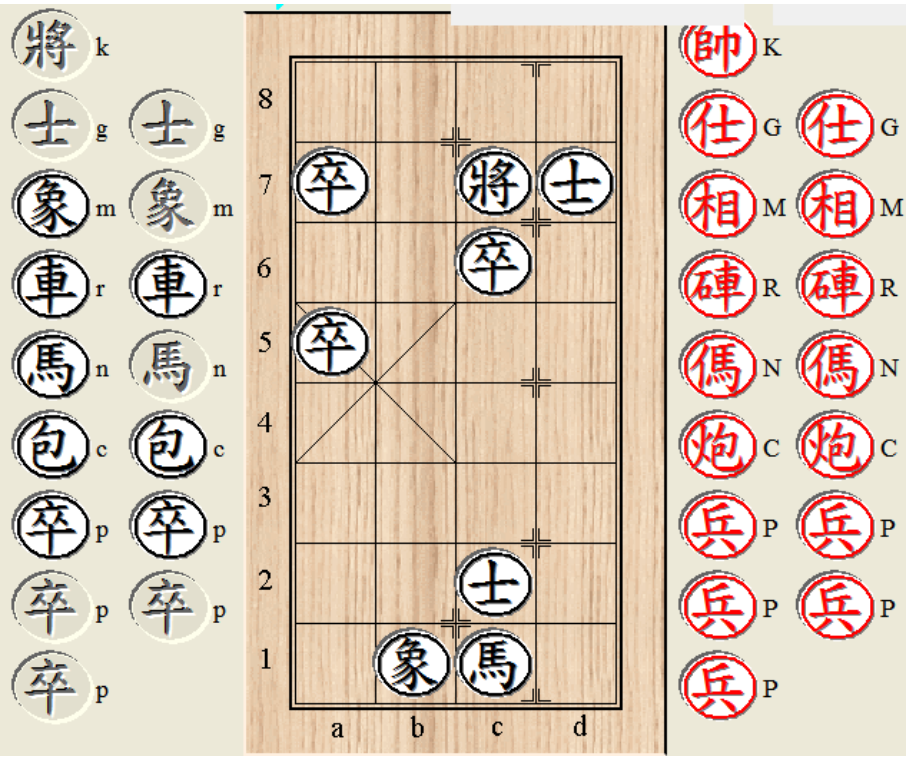


圖 7.6 DarkCraft 比賽獲勝盤面二

第二節 向上傳遞更新法

傳統暗棋程式的審局方法為：每當搜尋到葉節點時，將盤面上所有的子之子力分數加起來。若一個盤面搜尋時大部分的葉節點都沒有發生吃子狀況，它還是需要將盤面重新計分，同樣的加法會重複很多次，非常沒有效率，但是暗棋有一個特性：只有在吃子時審局分數才會改變，本研究以此特性為暗棋程式做了改良。原本的搜尋是在搜尋到葉節點時為盤面審局，如下所示：

```
if(depth ==0)
        return Evaluate();
```

新的方法如程式碼7.3:搜尋到葉節點時傳回0,搜尋途中若發生吃子狀況，就將該子的分數(weight[cap])加到遞迴時用到的 value 上面，並累積 value 的值再往上傳遞，這樣可以讓程式省掉很多不必要的計算，增加程式搜尋速度。

```
Search(int depth){
    if(depth ==0)
        return 0;                       //這邊改為零
    for(MoveLeft()){
        cap = NextMove();               //如果不是吃子步則 cap 為 0，
        if(cap!=0){                     //如果是，cap 為被吃子的編號。
            value += weight[cap];       //將被吃子的分數累積起來往上傳
        }
        value  −= Search(depth - 1);    //這裡是 value = value − Search(depth - 1);
    }
}
```

程式碼 7.3 程式重點修改部位，其他部位與第三章的 Nega-Max Search 相同。

要注意的地方為：由於使用加法累積，我方的子被吃時對我方不利，在 weight[ ]中我方的子為負分，對方的子為正分，和傳統審局的分數相反。

第三節 同分步處理

在搜尋的過程中很容易出現同分走步的狀況，以下為簡單的範例盤面：

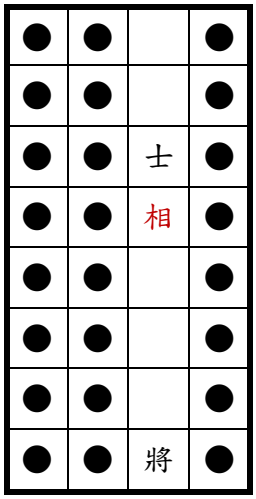


圖 7.7 同分走步範例盤面

假設我方為黑方，吃掉相可以獲得 900 分，在這個盤面裡，士只需走一步就可以吃掉相，而將必須走四步才吃得到相，如果沒有特殊處理的話，用哪顆子吃掉相都可以得到 900 分，但是一般 MoveGen( )產生走步時，我們都會先產生大子的走步，在這個範例中將的走步會在士的走步前面，而演算法中：

```
if(value > best){
        best = value;
}
```

這個判斷會讓程式選擇移動將，因為兩邊都是 900 分，所以程式不會選擇士的 900 分，這與我們策略裡「用最快的速度，將對手的子吃掉」相違背，我們要用最快的速度把對手的子吃掉才行。以這個盤面為例，如果當我們花四步移動將時，對手在相的旁邊翻出帥來，我們就較難吃掉相了。

要改善這種狀況只如程式碼 7.4 需改一行程式即可。

```
Search(int depth){
        if(cap != 0){
            value += ( weight[cap] + maxDepth – depth);      //改這行即可
        }
}
```

程式碼 7.4 同分步處理

程式碼 7.4 為第二節的 Search 使用同分步處理需要修改的部分，其他地方跟原本一樣。幫「weight[cap]」加上額外的分數「maxDepth – depth」，假設範例盤面搜尋八層，則 maxDepth 就是 8，士在第一層就吃掉相，depth 為 1，則用士吃可以額外得到 8-1=7 分，而將要在第四層才可以吃到相，則用將吃只能額外得到 8-4=4 分，這樣用比較淺層的士吃可以得到 907 分，而用比較深層的將吃可以得到 904 分，程式就會選擇使用較近的士去吃相了。

第四節 單向搜尋

經常會有我方子離對方太遠而搜尋不到的狀況，圖 7.8 為一個簡單的範例盤面：

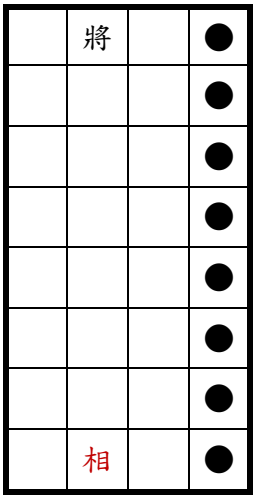


圖 7.8 單向搜尋範例盤面

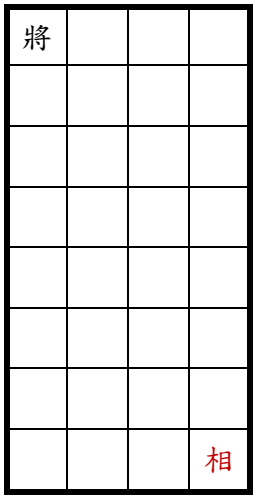


圖 7.9 吃不到的例子

如果程式的搜尋深度是十層，搜尋時黑方只能走五步，紅方也走五步，這個盤面黑方要走七步才能吃掉相(不考慮翻子狀況)，這樣不管怎麼搜尋，黑方都找不到能吃掉相的走法，而且黑方每個候選步的分數都是吃不到子的 0 分，原本能吃到相的將就不會朝著相的方向前進，也就無法吃到相了。

如果每個候選走步的分數都是 0 分的話，就可以使用單向搜尋法，擴大自己吃子的搜尋範圍。程式和原本的搜尋一模一樣，只是在遞迴呼叫時把「換邊」的動作拿掉，如果要搜尋十層而我方是黑方，就讓黑方連續走十步，這樣就可以解決範例盤面的問題了。

這個方法只能讓自己的大子在搜尋不到吃子步時，會有方向性的朝對方能被自己吃的子前進，不代表一定能吃到對方。圖 7.8 中，將跟相的曼哈頓距離為偶數，若黑先手一定吃不到相，但還是會朝向相前進，若黑後手就能吃到相。

第五節 寧靜搜尋

一般搜尋演算法的深度有限，在一定深度之後的狀況很有可能被忽略。例如我們搜尋深度為八層，最好的一手為 800 分，但如果我們走這步，對方在第九手時可以把我們的大子吃掉讓我們變 0 分，我們得到的 800 分就是錯的結果。

我們使用寧靜搜尋來解決這種問題，當搜尋到最深的地方時，如果還有吃子狀況發生，我們就繼續搜尋直到沒有吃子狀況為止，如程式碼 7.5 所示。

```
Search(int depth){
        if(cap != 0){                          //發生吃子狀況
            value += ( weight[cap] + maxDepth – depth);
            if(depth == (maxDepth-1))          //倒數第二層不減少搜尋層數
                value  −= Search(depth);
            else
                value  −= Search(depth - 1);   //減少搜尋層數
        }
}
```

程式碼 7.5 寧靜搜尋

程式碼 7.5 為第二節的 Search 使用寧靜搜尋需要修改的部分，其他地方跟原本一樣。如果搜尋到倒數第二層發生吃子狀況，則遞迴呼叫時不減少搜尋層數 depth，就可以額外多搜尋一層，達到寧靜搜尋的效果。

第八章 結論與未來方向

第一節　結論

BitBoard 能應用在多種不同的程式上，但是在有些遊戲中會出現特殊狀況或規則，例如本研究的炮位問題，這時就可以利用本研究的切割與合併的技巧來解決許多本來難以解決的狀況，這不只是解決單一問題的方法，也是一種新的思維，經過些許轉換，就能簡化不同問題。圖 8.1 為產生西洋棋城堡走步簡化範例，此問題原本是使用一種稱為 Magic Bits 的方法解決[5]，非常複雜。

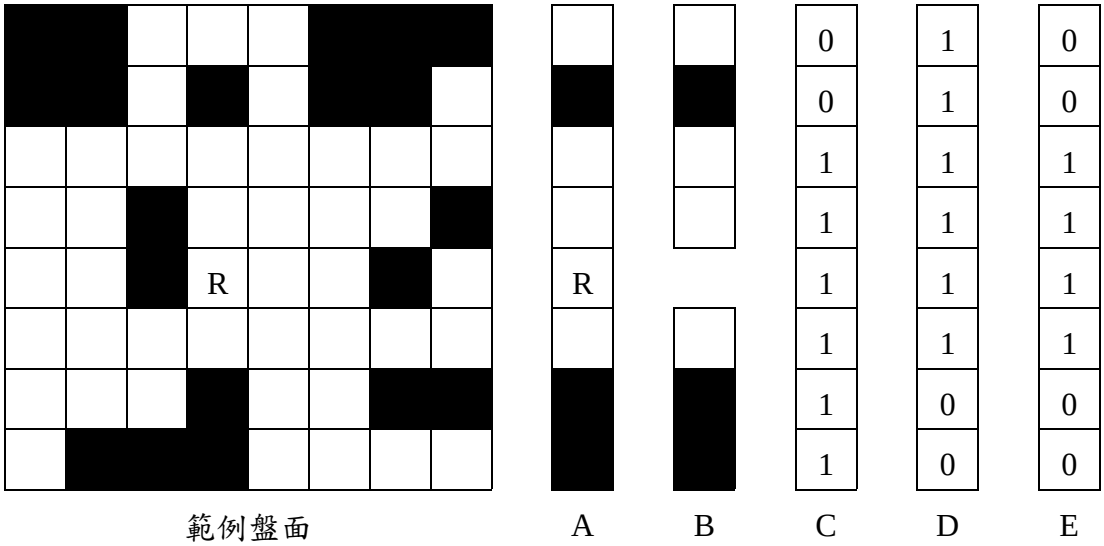


圖 8.1 簡化範例

範例盤面為一個西洋棋盤面，R 為城堡，黑色框框為任意一子，白色框框為空格，A、B、C、D、E 為各行的索引。

這裡以產生與城堡同一行的走步做為示範，同一列的走步可以用一樣的方法產生，和產生炮之走步的方法一樣，先將與城堡同一行的資料切割下來移到第一行，得到行 A，再將它分為上下兩段得到行 B，上半段中找出離城堡最近的黑色框框，本範例中為上面數下來第二個位置，則將從上面數下來第二個位置以上的位元全部填入 0(城堡不能走的位置)，第三個位置以下全部填入 1(城堡可以走的位置)得到行 C，行 B 中下半段也是找出離城堡最近的黑色框框，與上一個步驟相反，以上全部填 1，以下全部填 0 得到行 D，將行 C 與 C 行 D 做一次 AND 的運算就可以得到城堡在該行的走步了。

演算法部份，透過本研究所研發出來的暗棋程式 DarkCraft，棋力有很大的提升，在比賽中也有不錯的成績，這些演算法確有實效。

第二節　未來方向

目前電腦皆為 64 位元和 32 位元，許多遊戲的棋盤不是剛好有 64 格或 32 格。例如 19 路圍棋棋盤有 361 格，若要儲存 361 個位元就需要 6 組 64 位元，還會空下 23 個位元，要整合 6 組 64 位元也不容易。9 路圍棋有 81 格也不是 32 的倍數，像這類遊戲要使用 BitBoard 就需要克服更多問題或讓許多位元空著，這部分仍有很大的發展空間。

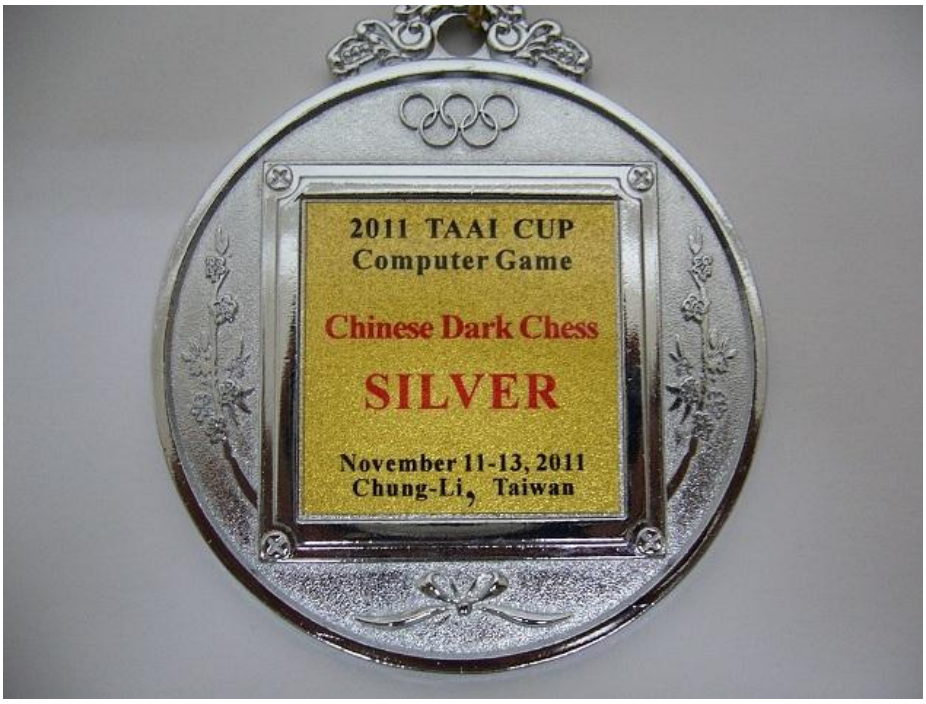
暗棋與其他遊戲最大的不同處在於，它非常容易和局，正式比賽中也常出現。本來大贏的盤面以和局收場，或要輸可是能拖到和局的盤面卻輸掉。由於這個遊戲有未翻子，加上棋子會循環相剋，就算知道剩下那些子沒翻開，也不知道他們到底在哪裡，所以要準確判斷局勢好壞非常困難。如果能想出完整判斷局勢好壞的方法，就可以在大贏但快和局的盤面翻子，以增加獲勝的機會。但在快輸的盤面不要翻子，一直拖到和局，也能結合翻子與走子的搜尋產生更多變化的下法。未來可以加入機器學習和動態審局配分再強化程式。

目前初步推測，選擇良好的翻子時機能解決上述問題，目前演算法翻子時機為「得到盤面資料先做走步搜尋，當目前無法吃到敵方子，且己方沒有子會被吃時才翻子」。若改成「得到盤面資料先做翻子搜尋，得到期望值分數，再做走步搜尋，並設計一個基準來比較哪個動作較好，再選擇接下來的動作」，當我們劣勢時，翻子分數可能會較低，就不會翻子，盡量走空步讓遊戲和局，優勢時，翻子分數就會較高，就會翻子讓遊戲不要和局，這樣能讓棋力大幅提升。

第九章 附錄

1. DarkCraft 比賽成績

The 16th Computer Olympiad 2011, 1st Prize



2011 TAAI CUP Computer Game, SILVER Prize

2. The 16th Computer Olympiad 與各對手比賽棋譜

Dark_chesser 後手持紅，和局。	Dark_chesser 後手持黑，和局。	Dark_chesser 先手持黑，對方黑勝。	
1. c6(c) c5(P)	1. c6(N) c4(g)	1. a1(p) a3(r)	34. c2(G) b5(g)
2. c8(P) c7(p)	2. a4(M) a2(c)	2. d1(P) c1(C)	35. c6-d6 b3-b4
3. c6-c8 c5-c6	3. a3(M) a2-a4	3. a2(n) c3(C)	36. b5-c5 d7(n)
4. c8-c6 c4(p)	4. a3-a4 a6(C)	4. b3(N) c3-a3	37. b6-b7 c7-d7
5. a5(r) c2(P)	5. a8(G) c8(c)	5. a2-a3 b3-a3	38. d8-d7 d3(p)
6. d8(m) a1(R)	6. c2(n) c8-a8	6. b2(M) c1-a1	39. b7-b6 c2-c3
7. b8(P) b7(C)	7. a6-a8 a7(P)	7. d4(P) a5(p)	40. b6-b5 c3-d3
8. d8-c8 a8(n)	8. d8(P) b1(G)	8. a6(M) a6-a5	41. b5-b4 d3-d4
9. c8-b8 d6(M)	9. a5(R) d1(r)	9. a8(K) b6(m)	42. b4-b3 d4-d3
10. b8-b7 d6-c6	10. d7(p) d7-d8	10. c8(c) b8(k)	43. b3-c3 d3-d2
11. c7-c8 c6-c5	11. a8-d8 b8(p)	11. c8-a8 a3-b3	44. c3-c2 d2-d3
12. b7-c7 c5-c4	12. a4-a3 b8-c8	12. a8-a5 a1-a5	45. c5-c4 d3-d2
13. c7-b7 c4-c5	13. a3-a4 c8-b8	13. b6-a6 b7(N)	46. c2-c1 d2-d3
14. b7-c7 c5-c4	14. a4-a3 b8-c8	14. b8-b7 a7(r)	47. c1-d1 d3-d4
15. c7-b7 c4-c5	15. a3-a4 c8-b8	15. a6-a5 c4(m)	48. c4-d4
16. b7-c7 c5-c4	16. a4-a3 b8-c8	16. c4-d4 c5(R)	
17. c7-b7 c4-c5	17. a3-a4 c8-b8	17. d4-c4 b4(P)	
18. b7-c7 c5-c4	18. a4-a3 b8-c8	18. c4-b4 d5(c)	
19. c7-b7 c4-c5	19. a3-a4 c8-b8	19. d5-d4 b3-c3	
20. b7-c7 c5-c4	20. a4-a3 b8-c8	20. d2(p) c5-d5	
21. c7-b7 c4-c5	21. a3-a4 c8-b8	21. b4-c4 d5-d4	
22. b7-c7 c5-c4	22. a4-a3 b8-c8	22. d2-d1 d4-d5	
23. c7-b7 c4-c5	23. a3-a4 c8-b8	23. c4-c3 d8(g)	
24. b7-c7 c5-c4	24. a4-a3 b8-c8	24. b7-b6 d6(P)	
25. c7-b7 c4-c5	25. a3-a4 c8-b8	25. c3-c4 a4(p)	
26. b7-c7 c5-c4	26. a4-a3 b8-c8	26. c4-d4 d5-c5	
27. c7-b7 c4-c5	27. a3-a4 c8-b8	27. d4-d5 c5-c4	
28. b7-c7 c5-c4	28. a4-a3 b8-c8	28. d5-d6 c4-b4	
29. c7-b7 c4-c5	29. a3-a4 c8-b8	29. c7(G) b4-a4	
30. b7-c7 c5-c4	30. a4-a3 b8-c8	30. a5-a4 c6(P)	
31. c7-b7 c4-c5	31. a3-a4 c8-b8	31. d6-c6 b2-b3	
32. b7-c7 c5-c4	32. a4-a3	32. b1(R) b1-c1	
33. c7-b7		33. a4-a5 c1-d1	

Dark_chesser 先手持黑，和局。	Diablo 先手持紅，對方紅勝。		Diablo 先手持紅，我方黑勝。
1. a1(p) a3(P)	1. b2(C) b3(g)	37. a2-a3 b7-c7	1. b2(P) b4(r)
2. c3(r) a4(g)	2. d2(p) b3-b2	38. c6-c7 b8-b5	2. d3(M) b5(P)
3. a4-a3 a6(R)	3. b5(C) b2-b3	39. d5-d6 d2-d3	3. c3(R) b4-b5
4. c6(C) c8(p)	4. b5-b3 c2(m)	40. a3-a4 d3-d4	4. b7(p) b8(c)
5. c7(M) c7-c8	5. d3(p) d4(P)	41. a4-b4 d4-d5	5. c8(K) b5-b4
6. a3-a4 a7(N)	6. b3-d3 c3(p)	42. b4-b5 d5-d6	6. a8(r) b4-b5
7. a4-a3 c4(R)	7. a3(N) c3-b3	43. c7-c8 c1(P)	7. d8(p) d8-c8
8. c3-c4 c8-c7	8. d1(k) b3-b2	44. b6(n) b6-c6	8. c7(C) d7(G)
9. a3-a4 c7-c8	9. d3-d1 d2-d3	45. c8-d8 d6-d5	9. c2(G) b5-b4
10. a4-a3 c8-c7	10. d5(p) c2-d2	46. c3-d3 d5-d6	10. c5(m) c5-b5
11. a3-a4 c7-c8	11. d1-d3 d2-d3	47. d3-d4 c6-c7	11. c4(c) c4-c2
12. a4-a3 c8-c7	12. d4-d5 d3-c3	48. d4-d5 d6-c6	12. c1(N) b5-c5
13. a3-a4 c7-c8	13. a2(R) c3-b3	49. a5(R) a7-a6	13. a6(p) c6(P)
14. a4-a3 c8-c7	14. a4(G) b2-c2	50. a5-a6 c7-b7	14. c1-c2 c5-c6
15. a3-a4 c7-c8	15. c7(p) b3-b2	51. d8-c8 d7(M)	15. c3-c4 b4-b5
16. a4-a3 c8-c7	16. b4(G) c2-d2	52. b5-b4 c6-b6	16. c2-c1 c6-c5
17. a3-a4 c7-c8	17. d5-d4 b7(P)	53. d7-c7 b7-b8	17. d3-c3 c5-c6
18. a4-a3 c8-c7	18. b7-c7 a6(r)	54. a6-a7 b6-a6	18. b2-c2 c6-c5
19. a3-a4 c7-c8	19. a3-b3 b2-a2	55. c7-b7 c4(g)	19. a4(M) c5-c6
20. a4-a3 c8-c7	20. b3-c3 a2-b2	56. b4-c4 b8-a8	20. d1(n) d1-c1
21. a3-a4 c7-c8	21. d4-d3 d2-d3	57. c8-b8 a6-a5	21. a1(m) b5-c5
22. a4-a3 c8-c7	22. c3-d3 d8(n)	58. c4-b4 a5-a6	22. c4-c5 c6-c5
23. a3-a4 c7-c8	23. b4-b3 b2-c2	59. b4-b5 a6-a7	23. a5(g) a5-a4
24. a4-a3 c8-c7	24. b3-c3 c2-d2	60. b7-a7	24. a3(n) a4-a5
25. a3-a4 c7-c8	25. d6(M) c5(P)		25. d4(k) a5-b5
26. a4-a3 c8-c7	26. d3-d4 b8(P)		26. d2(g) c5-c6
27. a3-a4 c7-c8	27. c3-d3 d2-c2		27. b3(C) a3-b3
28. a4-a3 c8-c7	28. d3-c3 c2-d2		28. d5(P) d4-d3
	29. d4-d5 a8(m)		29. c3-b3 d2-c2
	30. a4-b4 a8-b8		30. b3-a3 c6-c7
	31. b4-b5 b1(K)		31. d7-c7 d3-c3
	32. c6(c) a7(N)		32. c7-b7 c3-b3
	33. d6-c6 a6-a7		33. d6(P) b3-a3
	34. a1(r) b8-b7		34. d6-d7 b5-c5
	35. b1-a1 c8(c)		35. b7-b8 c5-d5
	36. a1-a2 c8-b8		36. a2(N) a3-a2

	Diablo 後手持紅，我方黑勝。		Diablo 後手持黑，對方逾時負。
37. b8-c8 d5-d6	1. c6(m) c3(g)	36. a8-b8 b1-a1	1. c6(R) d2(c)
38. c8-d8 a8-b8	2. c7(P) c8(R)	37. c2-b2 a2-a3	2. d3(P) d4(m)
39. a7(p) b8-b7	3. c6-c7 c5(P)	38. d8-c8 d7(G)	3. d1(k) d5(M)
40. d8-c8 d6-d7	4. c7-c8 a3(p)	39. c1-d1 a3-a2	4. d5-d4 c1(n)
41. b1(R) a1-b1	5. c8-c7 c1(G)	40. d6-c6 a2-a3	5. d4-d5 d7(K)
42. c8-b8 b7-c7	6. c7-c6 b1(n)	41. b3-a3 d7-c7	6. c5(N) d8(m)
43. b6(p) b6-c6	7. c6-c5 a5(K)	42. c8-c7 a1-a2	7. d7-d8 a6(p)
44. b8-c8 a2-a3	8. a4(p) a6(c)	43. b2-a2	8. d8-d7 a2(N)
45. c8-b8 a6-a5	9. a4-a5 a7(p)		9. d7-d8 b1(G)
46. b8-c8 a3-a4	10. d3(C) d5(N)		10. d8-d7 a5(p)
47. c8-b8 a4-b4	11. c3-d3 b6(p)		11. d7-d8 a7(g)
48. b8-a8 b4-b5	12. c5-d5 b7(N)		12. d8-d7 a4(p)
49. a8-b8 b5-b6	13. a5-a4 b7-a7		13. d7-d8 b8(p)
50. b8-c8 b6-b7	14. a6-a5 a7-a6		14. d8-d7 b3(r)
51. c8-c7 b7-c7	15. d6(P) a6-a5		15. d7-d8 b4(M)
	16. d5-d6 a5-a4		16. d8-d7 b7(G)
	17. d4(m) c1-b1		17. b7-b8 c2(C)
	18. b3(p) d1(M)		18. b4-b3 c1-c2
	19. c2(M) b1-c1		19. b3-b4 d1-c1
	20. c4(c) c1-b1		20. b4-a4 c1-b1
	21. d3-c3 b5(P)		21. a4-a5 a1(C)
	22. c3-c2 a4-a3		22. b2(p) b1-a1
	23. b6-b5 d8(k)		23. a2-b2 c2-c1
	24. d8-c8 a3-b3		24. c7(P) b6(P)
	25. c2-c3 d2(P)		25. b6-b7 a1-a2
	26. c3-b3 d1-c1		26. b2-b3 a2-b2
	27. b3-c3 b4(g)		27. b3-b4 a3(R)
	28. c8-d8 a8(r)		28. b4-a4 b2-a2
	29. d8-c8 d2-c2		29. a3-b3 a6-b6
	30. c3-c2 a1(n)		30. b3-b2 a7-a6
	31. c4-c1 b2(r)		31. b2-c2 d2-d1
	32. b2-b3 b1-a1		32. c2-d2 d1-d3
	33. c8-d8 a1-b1		33. d2-d3 b6-b7
	34. d8-c8 a2(C)		34. b8-b7 a6-a5
	35. c8-d8 b8(R)		35. a4-b4 a5-a4

36. b4-b3 a4-b4
37. c8(r) a8(P)
38. d7-d8 c8-b8
39. b7-b8 c4(n)
40. c3(c) b4-b3
41. d3-c3 b3-c3
42. c5-c4 b5(g)
43. d5-d4 c3-c4
44. d4-d3 c4-c3
45. d3-d2 c3-c2
46. d2-d1 c1-b1
47. b8-b7 c2-c1
48. d1-d2 c1-d1
49. d2-c2 d1-c1
50. c2-c3 c1-c2
51. c3-d3 c2-d2
52. d3-c3 d2-c2
53. c3-d3 c2-d2
54. d3-c3 b5-a5
55. d6(P) d2-c2
56. c3-c4 c2-c3
57. c4-d4 c3-d3
58. d4-c4 d3-c3
59. c4-d4 c3-d3
60. d4-c4 d3-c3
61. c4-d4 c3-d3
62. d4-c4 d3-c3
63. c4-d4 c3-d3
64. d4-c4 a2-a3
65. b7-b6

比賽說明：
比賽規則為雙循環，雙方各先手兩次、後手兩次。

抬頭說明：
範例：「Dark_chesser 後手持紅，和局」。
代表我方和 Dark_chesser 對戰，
對方後手持紅、我方先手持黑、此局和局。

棋譜說明：
棋盤編號：

8				
7				
6				
5				
4				
3				
2				
1				
	a	b	c	d

棋子代號：

K	G	M	R	N	C	P
帥	仕	相	俥	傌	炮	兵
k	g	m	r	n	c	p
將	士	象	車	馬	包	卒

範例：
36. b4-b3 a4-b4
為第 36 回合，每回合雙方各下一步。
先手方將 b4 的子移到 b3 位置
後手方將 a4 的子移到 b4 位置

37. c8(r) a8(P)
為第 37 回合先手方翻開位於 c8 的未翻子，翻出一隻 r(車)
後手方翻開位於 a8 的未翻子，翻出一隻 P(兵)

第十章 參考文獻

[1]謝曜安，電腦暗棋之設計及實作，國立台灣師範大學資工所碩士論文，2008。

[2]賴學誠，電腦暗棋程式與經驗法則之配合與實作，國立東華大學資工所碩士論文，2008。

[3]謝政孝，暗棋中棋種間食物鏈關係之探討與實作，國立臺灣師範大學資工所碩士論文，2010。

[4]勞永祥，電腦暗棋之人工智慧改良，國立臺灣師範大學資工所碩士論文，2011。

[5] WIKI Chess Programming 網站

http://chessprogramming.wikispaces.com/

[6]Game Tree by Yosen Lin

http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html

[7]維基百科

http://zh.wikipedia.org/